



Luca Cabibbo
Architettura
dei Sistemi
Software

Invocazione remota: REST

dispensa asw835
marzo 2021

*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**, fifth edition. Manning, 2019.
 - Chapter 6, **Creating REST services**
 - Chapter 7, **Consuming REST services**
- JSON
 - <https://www.json.org/>



- Obiettivi e argomenti

- Obiettivi
 - presentare l'invocazione remota tramite REST
- Argomenti
 - introduzione a REST
 - esempi
 - 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

- Dunque, 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 metodo del server con un'operazione HTTP relativa a un certo URI
 - (lato client) effettuare l'invocazione remota di un metodo del server – tramite una richiesta HTTP all'URI associato al metodo
 - questi metodi remoti 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 vengono scambiati su HTTP in modo opportuno – in genere 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 dei metodi remoti con **@RequestBody** e il risultato con **@ResponseBody**
 - sia i parametri che il risultato di un metodo remoto 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 nei 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 le operazioni HTTP – come GET, PUT, POST 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
 - DELETE – cancella l'intera collezione
 - 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
 - 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.rest.hello.domain;

import org.springframework.stereotype.Service;

@Service
public class HelloService {

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

}
```

- vogliamo esporre tale 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
 - l'operazione HTTP e il path
 - i parametri, con i loro tipi – 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.rest.hello.rest** con una classe **HelloRestController** che è il controller REST (di Spring Web MVC) per il servizio Hello



Server REST

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

```
package asw.rest.hello.rest;

import asw.rest.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** esemplifica 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 metodo all'operazione 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!**
- si noti l'adattamento (tipico di un adapter, in questo caso lato server) svolto dal metodo **sayHello**



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.rest.hello.client**
 - nel suo dominio definiamo un'interfaccia richiesta (una porta) **HelloServiceAdapter** per accedere al servizio Hello

```
package asw.rest.hello.client.domain;  
  
public interface HelloServiceAdapter {  
    public String sayHello(String name);  
}
```

- applicando l'architettura esagonale, va definito un outbound adapter per accedere al servizio Hello, che implementa questa interfaccia
 - definiamo il package **asw.rest.hello.client.hello.rest** con la classe **HelloServiceAdapterRestImpl**



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 dall'interfaccia/porta `HelloServiceAdapter` – e soprattutto che il client non dipende in alcun modo da REST

```
package asw.rest.hello.client.domain;

import ...;

@Component
public class HelloClientRunner implements CommandLineRunner {

    @Autowired
    private HelloServiceAdapter helloServiceAdapter;

    public void run(String[] args) {

        ... helloServiceAdapter.sayHello("Luca") ...
        ... helloServiceAdapter.sayHello("World") ...

    }

}
```

21

Invocazione remota: REST

Luca Cabibbo ASW



Client REST

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

```
package asw.rest.hello.client.hello.rest;

import asw.rest.hello.client.domain.HelloServiceAdapter;
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 HelloServiceAdapterRestImpl implements HelloServiceAdapter {

    ... vedi dopo ...

}
```

22

Invocazione remota: REST

Luca Cabibbo ASW



Client REST

- Anche nella classe **HelloServiceAdapterRestImpl** sono necessarie alcune definizioni preliminari

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

```
# application.properties
asw.rest.hello.service.uri=localhost:8080
```

```
private WebClient webClient;
```

```
public HelloServiceAdapterRestImpl(
    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 è **RestTemplate** – ma è in via di deprecazione



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;
}
```

- si noti l'adattamento (tipico di un adapter, in questo caso lato client) svolto dal metodo **sayHello**



- 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, essa 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 e per il client HTTP
 - un'invocazione remota può terminare con un'eccezione (utilizzando **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 GET non dovrebbero provocare cambiamenti di stato sul server – invece le operazioni POST in genere lo cambiano, e così pure le PUT e le 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



- 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
 - i ristoranti sono definiti come un’entità JPA **Restaurant** – con attributi **id**, **name** e **location**
 - la gestione dei ristoranti avviene tramite il servizio **RestaurantService**



Il servizio RestaurantService

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

```
package asw.efood.restaurant-service.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.restaurant.service.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.restaurant.service.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.restaurant.service.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`

```
package asw.efood.restaurant.service.api.rest;
import java.util.*;
public class GetRestaurantsResponse {
    private Collection<GetRestaurantResponse> restaurants;
    ... costruttori e metodi get, set e toString ...
}
```



Presentation model

- Definiamo un package `asw.efood.restaurant.service.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`

```
package asw.efood.restaurant.service.api.rest;
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.restaurantsevice.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, con il framework Spring, è possibile effettuare automaticamente la conversione tra queste classi e le strutture di dati JSON mostrate in precedenza, in entrambi i versi – utilizzando le librerie Jackson 2 per JSON



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.restaurantsevice.rest;

import asw.efood.restaurantsevice.domain.*;
import asw.efood.restaurantsevice.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, 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.restaurant-service.client`
 - nel package `asw.efood.restaurant-service.client.domain` va definita un'interfaccia richiesta (una porta) `RestaurantServiceAdapter` 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.restaurant-service.client.restaurant-service.rest` con la classe `RestaurantServiceAdapterRestImpl`



Client REST

- Lato client, l'interfaccia **RestaurantServiceAdapter**

```
package asw.efood.restaurant.service.client.domain;

import java.util.*;

public interface RestaurantServiceAdapter {

    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 ...

}
```

45

Invocazione remota: REST

Luca Cabibbo ASW



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 **RestaurantServiceAdapter** – e soprattutto che il client non dipende in alcun modo dall'adozione di REST

```
package asw.efood.restaurant.service.client.domain;

import ...;

@Component
public class RestaurantClientRunner implements CommandLineRunner {

    @Autowired
    private RestaurantServiceAdapter restaurantServiceAdapter;

    public void run(String[] args) {

        ... restaurantServiceAdapter.getRestaurant(1L) ...

    }

}
```

46

Invocazione remota: REST

Luca Cabibbo ASW



Client REST

- Lato client, la classe `RestaurantServiceAdapterRestImpl`

```
package asw.efood.restaurantService.client.restaurantService.rest;
import asw.efood.restaurantService.client.domain.*;
import asw.efood.restaurantService.api.rest.*;
import ...;

@Service
public class RestaurantServiceAdapterRestImpl
    implements RestaurantServiceAdapter {
    ... vedi dopo ...
}
```

- si noti, nell'implementazione della classe, l'adattamento tipico di un adapter (in questo caso 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 `RestaurantServiceAdapterRestImpl`

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

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

```
private WebClient webClient;

public RestaurantServiceAdapterRestImpl(
    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;  
}
```

51

Invocazione remota: REST

Luca Cabibbo ASW



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;  
}
```

52

Invocazione remota: REST

Luca Cabibbo ASW



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 ed 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



- Swagger UI

- **Swagger UI** (<https://swagger.io/tools/swagger-ui/>) consente di visualizzare e interagire con un servizio REST mediante una semplice interfaccia web generata automaticamente a partire dal servizio REST
 - con Spring Boot, nell'applicazione server, va utilizzata la dipendenza starter **io.springfox:springfox-boot-starter** e va definita una classe di configurazione per Swagger (che specifica i package in cui sono definiti i controller REST del servizio)
 - viene automaticamente generato 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



Swagger UI

The screenshot shows the Swagger UI interface for the 'restaurant-rest-controller' API. The page title is 'Api Documentation 1.0'. Below the title, there are links for 'Api Documentation', 'Terms of service', and 'Apache 2.0'. The main section is titled 'restaurant-rest-controller Restaurant Rest Controller'. It lists three endpoints: a GET endpoint for '/rest/restaurants' (getRestaurants), a POST endpoint for '/rest/restaurants' (createRestaurant), and a GET endpoint for '/rest/restaurants/{restaurantId}' (getRestaurantWith404). Below the endpoints, there is a 'Models' section with four expandable items: 'CreateRestaurantRequest', 'CreateRestaurantResponse', 'GetRestaurantResponse', and 'GetRestaurantsResponse'.

55

Invocazione remota: REST

Luca Cabibbo ASW



Swagger UI

The screenshot shows the Swagger UI interface for the 'restaurant-rest-controller' API, specifically the execution interface for the 'POST /rest/restaurants createRestaurant' endpoint. The 'Parameters' section is expanded, showing a table with columns 'Name' and 'Description'. The first row is 'request * required', with a description of 'REQUEST object (body)'. Below the table, there is a text area for the request body containing the JSON:

```
{  "location": "Roma",  "name": "Barfetto"}
```

. There are 'Cancel' buttons for the parameters section and the text area. Below the text area, there is a 'Parameter content type' dropdown menu set to 'application/json'. At the bottom, there is a large blue 'Execute' button. Below the 'Execute' button, there is a 'Responses' section with a 'Response content type' dropdown menu set to '*/'. At the very bottom, there is a table with columns 'Code' and 'Description'.

56

Invocazione remota: REST

Luca Cabibbo ASW



The screenshot shows the Swagger UI interface for a REST API endpoint. The 'Responses' section is expanded, showing the following details:

- Curl:** `curl -X POST "http://localhost:8080/rest/restaurants" -H "accept: */*" -H "Content-Type: application/json" -d '{"location": "Roma", "name": "Baffetto"}'`
- Request URL:** `http://localhost:8080/rest/restaurants`
- Server response:**
 - Code:** 200
 - Response body:** `{ "restaurantId": 5 }`
 - Response headers:**

```
connection: keep-alive
content-type: application/json
date: Thu, 08 Apr 2021 10:16:25 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```
- Responses table:**

Code	Description
200	OK

Example Value | Model

```
{ "restaurantId": 0 }
```



* 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 lato server (un controller REST) e di un adapter 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)