Programmazione Web - Lezione 12: ReST, XML Avanzato, Web Services

12.1 ReST

ReST è l'acronimo di **Re**presentational **S**tate **T**ransfer. È importante chiarire subito cosa sia: ReST è uno **stile architetturale**, un insieme di principi e vincoli per progettare sistemi distribuiti, in particolare applicazioni web. Non è uno standard o un protocollo specifico, ma piuttosto un modo di organizzare l'architettura con cui i servizi vengono definiti e interagiscono. Spesso, viene erroneamente confuso con qualsiasi Web Service che sia semplicemente richiamabile tramite il protocollo HTTP.

Per comprendere meglio, chiariamo cosa si intende per Web Service. Un Web Service è fondamentalmente un servizio orientato ad altre applicazioni, che espone delle API (Application Programming Interface). Queste API vengono utilizzate da altre applicazioni per ottenere servizi, scambiando dati (fornendoli o ricevendoli). Poiché la comunicazione avviene tipicamente tramite richieste HTTP, questi servizi prendono comunemente il nome di "Web Service".

La relazione tra Servizi Web e ReST è che un servizio web *può* essere sviluppato seguendo lo stile architetturale ReST, ma può anche essere realizzato *senza* aderire a tali principi, come nel caso dei tradizionali servizi basati su SOAP/WSDL. Quando un servizio o un'architettura aderiscono ai vincoli e ai principi definiti dallo stile ReST, vengono definiti **ReSTful**. Questi principi si concentrano in modo particolare sulle modalità di trasmissione e manipolazione dei dati e sull'identificazione delle risorse.

I principi fondamentali di ReST includono:

1. **Client-Server:** L'architettura ReST impone una netta separazione tra client e server. Il client è responsabile dell'interfaccia utente e dell'inizializzazione delle richieste, mentre il server gestisce le risorse, elabora le richieste e invia le risposte. Questa separazione delle preoccupazioni (Separation of Concerns) migliora la portabilità del client e la scalabilità del server, consentendo loro di evolvere in modo indipendente.
2. **Stateless (Senza Stato):** La comunicazione tra client e server deve essere priva di stato. Ciò significa che ogni richiesta inviata dal client deve contenere tutte le informazioni necessarie al server per comprenderla ed elaborarla, senza fare affidamento su un contesto di sessione memorizzato sul server tra una richiesta e l'altra. Se è necessario mantenere uno stato (come l'autenticazione), questo deve essere gestito dal client e inviato con ogni richiesta. Questo

approccio semplifica il server, ne migliora l'affidabilità e la scalabilità, poiché il lavoro del server si esaurisce completamente con la risposta alla singola richiesta.

3. **Cacheable (Memorizzabile nella Cache):** Le risposte fornite dal server dovrebbero indicare se sono memorizzabili nella cache o meno. Se una risposta è cacheabile, un client o un intermediario (come un proxy) possono riutilizzarla per richieste successive identiche, migliorando significativamente le prestazioni e riducendo il carico sul server. Questo si basa sul fatto che, per una data risorsa, richieste identiche in momenti ravvicinati dovrebbero produrre la stessa risposta se lo stato della risorsa non è cambiato.
4. **Uniform Interface (Interfaccia Uniforme):** Questo è forse il principio più distintivo di ReST e mira a semplificare e disaccoppiare l'architettura. Si basa su diversi sotto-vincoli. Innanzitutto, le **risorse** (concetti come un utente, un prodotto, un ordine) vengono identificate univocamente tramite URI. I client interagiscono con queste risorse manipolando le loro **rappresentazioni** (ad esempio, un documento XML o JSON che descrive la risorsa). Una singola risorsa può avere molteplici rappresentazioni, e client e server negoziano quale utilizzare. I messaggi scambiati sono **auto-descrittivi**, contenendo informazioni sufficienti per essere elaborati (come il tipo di media della rappresentazione). Infine, il principio **HATEOAS (Hypermedia as the Engine of Application State)** stabilisce che il client naviga lo stato dell'applicazione seguendo i link presenti nelle rappresentazioni ricevute dal server. Ad esempio, la rappresentazione XML di un album potrebbe contenere elementi `<link>` con attributi `rel` (che descrive la relazione, es. "acquisto" o "artista") e `href` (l'URI per eseguire l'azione), guidando il client sulle azioni possibili senza che questo debba conoscerle a priori.

In sintesi, ReST si chiama **Representational State Transfer** perché il server trasferisce al client una **rappresentazione** dello **stato** corrente di una risorsa. Questa rappresentazione include spesso link ipermediali (hypermedia) che guidano il client verso le possibili azioni successive. L'uso di standard come URI e HTTP garantisce l'**uniformità** dell'interfaccia.

L'approccio ReST si rivela particolarmente adatto per i sistemi basati su **microservizi**. Ogni microservizio può gestire un insieme di risorse, esponendole tramite un'API ReSTful. Le rappresentazioni fornite possono includere link che puntano ad azioni gestite dallo stesso microservizio o da altri. Un componente come un *dispatcher* o un *API Gateway* può quindi instradare le richieste in entrata al servizio appropriato basandosi sull'URI, favorendo un'alta **componibilità** e disaccoppiamento.

Riguardo all'implementazione pratica di HATEOAS, sorge la domanda se usare **XML o JSON**. XML possiede un concetto abbastanza diffuso per i link, spesso rappresentato dall'elemento `<link rel="..." href="..."/>` . JSON, invece, non ha un meccanismo nativo standardizzato. Tuttavia, esistono convenzioni come **HAL (Hypertext Application Language)**. In un approccio simile ad HAL, un oggetto JSON può includere un campo speciale, ad esempio `"_links"` , contenente a sua volta un

oggetto dove le chiavi rappresentano le relazioni (come `rel` in XML) e i valori specificano l'URI e potenzialmente altre informazioni sul link.

Per approfondire ReST, si può consultare il riferimento: https://italiancoders.it/introduzione-a-rest/

12.2 Namespace in XML

Quando si lavora con XML, specialmente combinando vocabolari provenienti da fonti diverse, può sorgere il problema delle collisioni di nomi: elementi o attributi con lo stesso nome potrebbero avere significati differenti. Per risolvere questa ambiguità, XML introduce il concetto di **Namespace (Spazio dei Nomi)**. Un namespace raggruppa una serie di nomi (di elementi e attributi) sotto un identificatore univoco, tipicamente un URI. Questo concetto è analogo ai namespace in C++ o ai package in Java.

Per indicare che un elemento o attributo appartiene a un determinato namespace, si usa solitamente un **prefisso**, una breve stringa seguita da due punti (`:`), anteposta al nome locale (es., `soap:Envelope`). Tuttavia, questi prefissi devono essere **definiti** associandoli all'URI del namespace corrispondente. Questa associazione avviene tramite un attributo speciale `xmlns:prefisso="URI_del_namespace"` , ad esempio `xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"` . Questa dichiarazione rende il prefisso `soap` un alias per l'URI specificato all'interno dell'elemento in cui è dichiarato e per tutti i suoi discendenti.

È fondamentale capire che l'**URI** associato al namespace serve principalmente come **identificatore univoco** e non deve necessariamente puntare a una risorsa web scaricabile. I processori XML utilizzano questo URI per riconoscere elementi appartenenti a vocabolari specifici. Se un processore riconosce l'URI associato a un prefisso, può interpretare correttamente gli elementi e gli attributi di quel namespace; altrimenti, potrebbe ignorarli o segnalare un errore.

L'uso dei namespace è cruciale perché permette di **integrare elementi da vocabolari diversi** nello stesso documento senza ambiguità, cosa difficile da gestire con le vecchie DTD. Inoltre, consente di dichiarare esplicitamente a quale specifica fa riferimento una parte del documento.

Un esempio pratico dell'uso dei namespace si trova nel protocollo **SOAP (Simple Object Application Protocol)**, uno standard W3C per lo scambio di messaggi XML tra sistemi. Un messaggio SOAP è un documento XML strutturato con un elemento radice `Envelope` (tipicamente con prefisso `soap:`), che contiene un `Header` opzionale e un `Body` obbligatorio. Il `Body` trasporta il payload effettivo della comunicazione, ossia i dati applicativi. È importante notare che, mentre l'envelope SOAP appartiene al namespace SOAP, il contenuto del `Body` appartiene tipicamente a un **altro namespace**, specifico

dell'applicazione o del servizio invocato. SOAP agisce come una busta neutra rispetto al suo contenuto.

Consideriamo un esempio di messaggio SOAP:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://magazzino.example.com/ws">
      <getProductDetailsResult>
        <productName>Matita</productName>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

Qui, `Envelope` e `Body` usano il prefisso `soap`. All'interno del `Body`, l'elemento `getProductDetailsResponse` e i suoi figli appartengono a un namespace differente, identificato dall'URI `http://magazzino.example.com/ws`. Notare l'uso di `xmlns="..."` senza prefisso sull'elemento `getProductDetailsResponse`: questo definisce un **namespace di default** per quell'elemento e per tutti i suoi discendenti che non hanno un prefisso esplicito.

Ricapitolando il flusso SOAP: il server destinatario riceve il messaggio XML. Un gestore SOAP processa l'envelope, riconoscendo gli elementi del namespace SOAP. Estrae quindi il payload dal `Body` e lo passa al componente applicativo appropriato, il quale è in grado di interpretare e processare quel payload perché ne conosce il namespace specifico (es. `http://magazzino.example.com/ws`).

12.3 XML Schema

Per definire in modo preciso la struttura, il contenuto e i tipi di dati dei documenti XML, è stato introdotto **XML Schema**, noto anche come **XSD (XML Schema Definition)**. XSD è un linguaggio basato su XML che rappresenta un'alternativa molto più potente e flessibile rispetto alle tradizionali DTD. Tra i suoi vantaggi principali vi sono un ricco sistema di **tipi di dati** predefiniti (stringhe, numeri, date, booleani, ecc.) e la possibilità di definire tipi personalizzati, il pieno supporto per i **namespace** e meccanismi avanzati per il **riuso** di definizioni e la creazione di strutture complesse. Data la vastità dello standard XSD, ne esamineremo solo gli aspetti fondamentali.

Un documento XSD è esso stesso un file XML, il cui elemento radice è `<xs:schema>`, dove il prefisso `xs` è convenzionalmente associato al namespace di XML Schema

(`http://www.w3.org/2001/XMLSchema`). All'interno di `<xs:schema>` vengono definite le regole per i documenti XML che si vogliono validare.

Per definire un **elemento semplice**, cioè un elemento che contiene solo testo, si usa `<xs:element>` specificando l'attributo `name` (il nome dell'elemento XML) e `type` (il tipo di dato del contenuto, es. `xs:string` , `xs:integer` , `xs:date`). Ad esempio, `<xs:element name="title" type="xs:string"/>` definisce un elemento `<title>` che deve contenere testo.

Per definire un **elemento complesso**, ossia un elemento che può contenere elementi figli e/o attributi, si usa `<xs:element>` contenente un `<xs:complexType>` . All'interno di `<xs:complexType>` si specifica la struttura interna. Per ordinare gli elementi figli si usano i *compositor*: `<xs:sequence>` (ordine fisso), `<xs:all>` (tutti presenti, ordine qualsiasi) o `<xs:choice>` (solo uno tra quelli elencati). La **molteplicità** di un elemento o di un compositor è controllata dagli attributi `minOccurs` (default 1) e `maxOccurs` (default 1, può essere `unbounded` per infinito).

Gli **attributi** vengono definiti all'interno di `<xs:complexType>` usando `<xs:attribute>` , specificando `name` , `type` e `use` (`"optional"` o `"required"`).

XSD promuove il **riuso**: è possibile definire elementi, attributi e tipi (semplici o complessi) a livello globale (come figli diretti di `<xs:schema>`) e poi riferirsi ad essi in altri punti dello schema usando l'attributo `ref` (per elementi e attributi) o usandoli come base per altre definizioni. XSD permette anche di definire gerarchie di tipi tramite estensione o restrizione.

Un documento XML è considerato **"Valido"** rispetto a un XSD se è ben formato e rispetta tutte le regole (struttura, tipi di dati, occorrenze) definite nello schema. La verifica della validità viene effettuata da un **Parser Validante XML Schema**. Un esempio pratico di file XSD è contenuto nel file `XML_Schema.pdf` menzionato nelle slide.

12.4 WSDL

WSDL (Web Service Definition Language) è un linguaggio basato su XML utilizzato per **definire formalmente un Web Service**. Il suo scopo principale è descrivere *cosa* fa il servizio (le operazioni offerte), *come* accedervi (protocolli e formati) e *quali* dati scambia (i messaggi e i loro tipi).

Questa definizione formale facilita enormemente lo **scambio di dati** e l'interoperabilità tra sistemi diversi. La conseguenza più importante è lo sviluppo di **librerie e strumenti (tooling)** che automatizzano gran parte del processo di comunicazione. Partendo da un file WSDL, questi strumenti possono generare codice **client (stub)** che semplifica l'invocazione del servizio, e viceversa, possono generare il file WSDL partendo dal codice sorgente di un servizio esistente o generare codice **server (skeleton)**.

Immaginiamo uno scenario tipico: un sistema S2 vuole esporre un servizio. Definisce le sue strutture dati e usa un tool per generare automaticamente un file WSDL che descrive il servizio. Questo WSDL viene inviato a un sistema S1 che vuole consumare il servizio. S1 usa a sua volta un tool WSDL per generare, nel proprio linguaggio, le classi corrispondenti alle strutture dati descritte nel WSDL e un client stub. Gli sviluppatori di S1 utilizzano queste classi generate per preparare i dati e poi invocano semplicemente un metodo sullo stub. Lo stub, guidato dal WSDL, si occupa automaticamente di serializzare i dati nel formato corretto (spesso XML dentro SOAP), inviare la richiesta via HTTP all'indirizzo specificato nel WSDL, ricevere la risposta, deserializzarla e restituire il risultato al codice chiamante. Un processo analogo avviene sul lato server (S2) per ricevere la richiesta, deserializzarla e processarla.

Anche se non analizzeremo la struttura interna dettagliata del WSDL, è importante capire le tecnologie su cui si basa. Un file WSDL è un documento **XML** che utilizza specifici namespace WSDL. Per definire i tipi di dati dei messaggi scambiati, WSDL si appoggia pesantemente su **XML Schema (XSD)**, adottandone quindi la visione a oggetti e il sistema dei tipi. Il protocollo di trasporto più comune specificato nei WSDL è **HTTP** (o HTTPS). Infine, molto frequentemente, i WSDL descrivono servizi che utilizzano **SOAP** per incapsulare i messaggi XML inviati via HTTP; il WSDL specifica come mappare le operazioni del servizio ai messaggi SOAP e come serializzare i dati XSD all'interno dell'envelope SOAP.

I **vantaggi** di questo ecosistema (WSDL/XSD/SOAP/HTTP) risiedono nell'**automazione**: gran parte della complessità della comunicazione è gestita dagli strumenti. Gli sviluppatori possono così concentrarsi sulla logica di business, lavorando con strutture dati nel loro linguaggio preferito, senza doversi preoccupare dei dettagli di basso livello dei protocolli o della serializzazione XML. La generazione automatica del WSDL e del codice client/server riduce inoltre i tempi di sviluppo e il rischio di errori.