POLITECNICO DI MILANO
V Facoltà di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica

Adattamento di servizi SOAP e REST: Progettazione e sviluppo di una proposta di soluzione

Relatore: Elisabetta Di Nitto
Correlatore: Gianluca Ripa

Tesi di Laurea di:
Alessandro MARASCO
matr. 682463

Anno Accademico 2009-2010

Abstract

Negli ultimi anni sono nati numerosi servizi Web che implementano un nuovo tipo di approccio che rispecchiano l'insieme dei principi denominati REST. Nel creare composizioni di servizi, si necessita spesso di eseguire una serie di invocazioni verso servizi di entrambe le tecnologie sia SOAP che REST. Questa tesi si pone l'obbiettivo di fornire un'approccio all'adattamento delle singole invocazioni tra servizi Web di diversa tipologia a livello di esecuzione in maniera trasparente rispetto alla composizione. Viene introdotta un'analisi sulle caratteristiche della nuova tipologia identificando le differenze con un servizio SOAP, che permetterà di esporre una soluzione basato sull'utilizzo di notazioni semantiche all'interno delle descrizioni dei servizi. Verrà introdotto un'implementazione di un prototipo che segue l'approccio proposto.

INDICE

Elenco delle figure

CAPITOLO 1

INTRODUZIONE

Il Web è sempre più presente nella nostra vita quotidiana, diventando quasi uno strumento fondamentale. La sua evoluzione è stata aiutata dall'utilizzo di strumenti tecnologici che hanno permesso di mettere a disposizione sempre più facilmente la reperibilità delle informazioni in esso contenuto.
Ad oggi si parla di Web 2.0 che ha, come caratteristica principale,la trasformazione dell'interazione tra sito-utente, introducendo il cambiamento dell'approccio con il quale gli utenti si rivolgono al Web, che passa fondamentalmente dalla semplice consultazione alla possibilità di contribuire popolando e alimentando il Web con propri contenuti spesso creati in cooperazione tra essi.
Questo tipo di approccio ha facilitato lo svolgere di determinate operazioni comuni come quello di effettuare acquisti di qualsiasi tipo e consultare servizi informativi di uso generale.
Una tecnologia che sta prendendo piedi in questi anni, segnatamente orientata ad affrontare le nuove problematiche e sfide del Web di oggi, è quella orientata ai servizi.

> Per **servizio** si intende "una risorsa astratta che rappresenta la capacità di eseguire dei compiti che formano un insieme coerente di funzionalità dal punto di vista del fornitore e utilizzatore del servizio".

> Per **Web Service** "*WS*" si intende "l'implementazione software di un servizio". I servizi Web sono una tecnologia basata su Xml e sono progettati per supportare l'interoperabilità e interazione tra macchine attraverso una rete.

La tecnologia Web Services, spesso nel documento chiamata semplicemente "servizi" o "servizi Web", e l'utilizzo di architetture orientate ai servizi (SOA), sembra ottenere un enorme successo.

Questo nuovo approccio alla rete Web consente di erogare informazioni disaccoppiate dalla rappresentazione grafica. I servizi sono dinamici, il risultato dell'elaborazione scaturita dalla richiesta può risultare diversa in base al contesto e/o a determinati criteri.
I Web Service verranno descritti più avanti nel capitolo 2, insieme a tecnologie, vantaggi e problematiche che derivano dal loro uso.
Il Web 2.0 ha introdotto nuovi strumenti e nuovi approcci al web, passando dalla semplice consultazione (seppure supportata da efficienti strumenti di ricerca, selezione e aggregazione) alla possibilità di contribuire, creando propri contenuti spesso anche in cooperazione tra essi.
Questa "cooperazione" ha investito anche i web service, modificandone l'uso tradizionale e introducendo nuove tipologie come i servizi REST. L'uso comune di Web Service, come descritti più approfonditamente nel prossimo capitolo, consente di far comunicare diversi sistemi tra loro rendendo l'iterazione indipendente dalla tecnologia e piattaforma utilizzata.
Se si pongono a confronto due servizi web che lavorano sullo stesso dominio di informazioni, si possono presentare molte diversità tra loro, anche se la creazione si basa sull'utilizzo di linguaggi standard che ne permettono una descrizione sulle modalità di utilizzo.
All'interno del mondo internet, esistono molti servizi, ma la loro eterogeneità rispetto alla realizzazione e una scarsa descrizione, portano ad una difficile comprensione da parte dell'utente sull'utilizzo di esso, e rende ancora più complessa una possibile comunicazione e scambio di dati tra i vari servizi effettuati da agenti autonomi.
Il lavoro descritto nel documento, vuole essere un contributo ad un progetto europeo chiamato *SOA4All* [36] e che ha come obiettivo principale quello di rendere i servizi web accessibili a tutti, sia ad utenti esperti quali sviluppatori, sia agli stessi utilizzatori. Per fare questo si cerca di analizzare e progettare nuove strade che permettano di semplificare la fruibilità e l'accessibilità dei servizi.
Il risultato del progetto SOA4All sarà una infrastruttura software che permetterà di integrare SOA (Service-Oriented Architecture) e quattro componenti rivoluzionari (il Web, le tecnologie context-aware, Web 2.0 e il Semantic Web) in un dominio coerente e indipendente dalla piattaforma di servizi.

1.1 Obiettivo

L'utilizzo e la diffusione delle nuove tipologie di servizi Web all'interno di una composizione di servizi, che generalmente viene descritta mediante un BPEL, introduce una maggior dinamicità tra i servizi.
In una composizione di servizi potremmo avere la necessità di dover combinare servizi di entrambe le tecnologie SOAP e REST in maniera dinamica e trasparente dal punto di vista dell'invocazione. Durante la modellazione del processo, vogliamo migliorare la disponi-

bilità dei servizi e l'affidabilità dell'intero processo indicando dei web service alternativi da invocare nel caso di fallimento da parte di quelli invocati.
L'obbiettivo principale di questo lavoro e il contributo concreto che si vuole dare al progetto SOA4All, è quello di ottenere una soluzione che permetta di adattare richieste e risposte tra servizi web anche di tecnologia diversa (REST e SOAP) arricchendo e inserendo nuovi meccanismi e concetti atti a descrivere i diversi servizi e a diminuire le difficoltà di comunicazione che nascono tra Web Service eterogenei.
Molto spesso, i Web Service che vengono invocati possono non rispondere a causa di anomalie dell'application server o per guasti sulla rete. Prendiamo il caso che l'utente *User* voglia effettuare una richiesta verso un servizio che fornisca "Previsioni Meteo", la prima operazione che deve effettuare è quella di ricercare un servizio che risponda ai suoi requisiti (questa operazione, nell'esempio riportato, viene effettuato sul *Discovery Agent*).

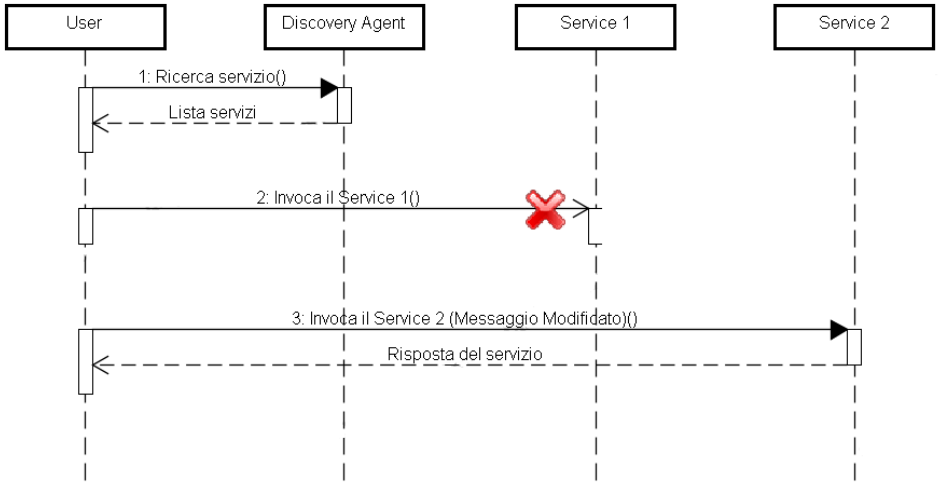


Figura 1.1: Caso d'uso: Invocazione di un servizio classico

La ricerca produrrà come risultato una serie di servizi tra cui *Service 1* e *Service 2*. *User* effettuerà una invocazione al servizio *Service 1*. L'invocazione prevede di fornire in input le coordinate geografiche della località in cui si vuole sapere le previsioni. Supponiamo che l'invocazione non vada a buon fine e venga restituito un messaggio di errore. L'utente, passerebbe al servizio successivo che il processo di ricerca gli ha fornito. Il *Service 2* ha le stesse caratteristiche e richiede gli stessi parametri solo che il messaggio deve essere composto in modo diverso da quello precedente. In questo caso il secondo servizio è raggiungibile e si ottiene il risultato del processo che è stato scatenato dalla richiesta. Lo scenario mostra come l'utente nei due servizi anche se sono molto simili a livello di informazioni di input e di finalità, ha dovuto modificare il messaggio di invocazione per poter ottenere una risposta da un servizio attivo, ma questo nella realtà non può avvenire in quanto i servizi vengono invocati da agenti autonomi e quindi non è possibile che un calcolatore possa automaticamente convertire un messaggio in un altro se non viene istrui-

to. Un altro problema che può essere sollevato si pone se anche il secondo servizio non rispondesse, allora bisognerebbe andare per tentativi e questo richiederebbe di modificare il messaggio e non solo, dato che ogni servizio risponde in maniera diversa e con formati eterogenei.
Proviamo ora a riprendere questo esempio e inseriamo tra l'utente e il mondo dei servizi un altro attore che chiamiamo *Service Adapter*.

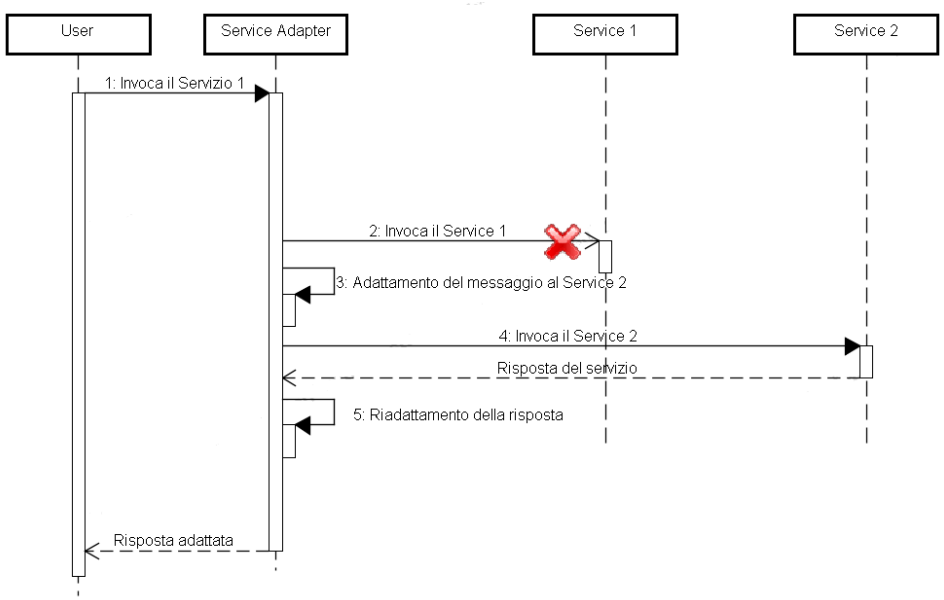


Figura 1.2: Caso d'uso: Invocazione di un servizio tramite Service Adapter

A questo punto l'utente effettua l'invocazione al *Service Adapter* invece che al *Service 1* con lo stesso messaggio che avrebbe inviato al servizio. Il *Service Adapter* effettua l'invocazione al *Service 1* e se non fosse raggiungibile, questo attore adatta automaticamente il messaggio ricevuto dall'utente ad un messaggio per la richiesta del *Service 2* che e sua volta invocato il secondo servizio, prende la risposta e la converte nello stesso formato di risposta del primo servizio in quanto l'utente si aspetta che sia il primo servizio a rispondere e non il secondo.
Nel nostro esempio, il *Service Adapter* effettua un'adattamento tra servizi e rende trasparente questo processo all'utente.
Questo adattamento, comporta notevoli problematiche da analizzare perchè l'eterogeneità non si ferma solo alla struttura dei singoli messaggi ma anche alla diversa tecnologia su cui i servizi vengono implementati.
Questo documento è un'estensione al lavoro realizzato da Teodoro De Giorgio nella sua tesi dal titolo "Il problema dell'adattamento nella composizione dei servizi: progettazione e sviluppo di una proposta di soluzione" [16] e si basa sull'articolo [9] che affronta la tematica di adattare due servizi implementati su protocollo SOAP. Noi andremo ad ag-

giungere prima l'adattamento tra servizi RESTful e dopo andremo ad adattare servizi tra tecnologie diverse RESTful su SOAP e SOAP su RESTful.

1.2 Struttura del documento

Il documento è organizzato nel seguente modo:

Capitolo 1 - Introduzione

Breve introduzione dei motivi su cui si basa questo lavoro, e la descrizione degli obiettivi che si vogliono raggiungere.

Capitolo 2 - Stato dell'Arte

Una panoramica sui servizi Web, sulle diverse tipologia soffermandosi sulla tipologia di servizi RESTful, descrivendone composizioni, vantaggi e svantaggi nell'utilizzo di questa tecnologia in confronto ad un'altra mediante esempi.
Introduzione di alcuni linguaggi che consentono di descrivere le strutture dei servizi sia a livello sintattico, descrivendo le strutture e i formati dei messaggi che vengono scambiati durante la comunicazione, sia a livello semantico. Alcuni verranno descritti in maniera approfondita, altre solamente accennate.

Capitolo 3 - Approccio Proposto

Verranno discussi i risultati degli studi effettuati per la realizzazione dell'obiettivo descritto nel documento, ed esposte le limitazioni attuate nell'implementazione della parte software.

Capitolo 4 - Progettazione e Implementazione

Verrà descritta l'architettura proposta ed esplicate le semplificazioni adottate nell'implementazione del lavoro.

Capitolo 5 - Conclusione

Si troveranno alcune considerazioni personali e alcuni spunti per sviluppi futuri dell'approccio proposto.

CAPITOLO 2

STATO DELL'ARTE

2.1 Servizi Web

Con l'integrazione di linguaggi di programmazione all'interno dei web server che generano codice HTML dinamicamente in base a determinati criteri, il web poggia su un modello di tipo *human-centric* dove l'utente umano è l'attore principale, il quale avvia il processo di richiesta ed esecuzione della pagina. Questo rappresenta anche il principale modello in cui la maggior parte del Web opera oggi.

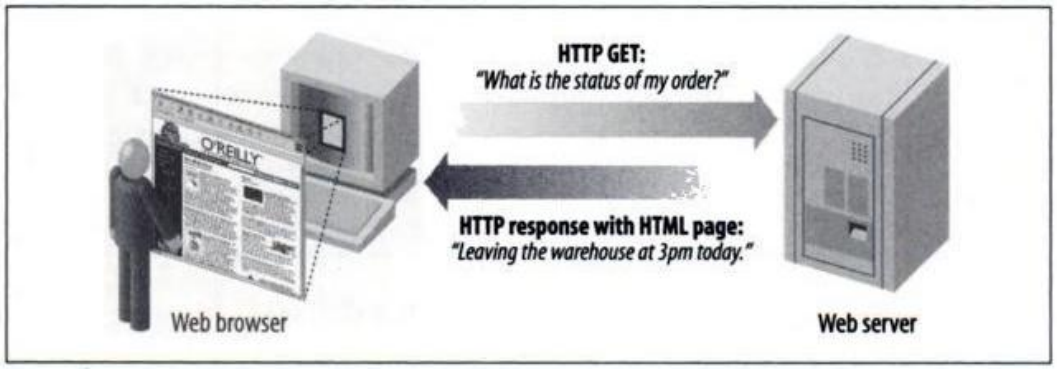


Figura 2.1: Esempio di modello *human-centric*

Lo sviluppo dei server Web che li hanno portati ad essere dei veri Application Server, ha spostato la concezione del web dal modello centrato sull'utente, verso un modello *application-centric*, dove la comunicazione avviene direttamente tra applicazioni, portando cosi non solo ad uno scambio di informazioni, ma anche ad una condivisione di risorse computazionali che vengono utilizzate a richiesta.
Questo modello è rappresentato da una architettura denominata SOA (Service-Oriented

Architecture)[19] e la sua implementazione più diffusa ad oggi, include tutte quelle tecnologie che vengono classificate come Web Service.
I Web Service vengono visti come soluzione tecnologica adatta all'interoperabilità dei sistemi poichè con l'introduzione di diversi linguaggi di programmazione orientati al web, le Web application sono sempre più dipendenti dalla piattaforma con cui vengono sviluppati. Va altresì sottolineato che il ruolo dei Web Service non si limita solo a un discorso di interoperabilità, bensì aiuta a superare la limitazione della "dipendenza" permettendo di descrivere nuovi servizi realizzati ad hoc, sempre però con l'intento di fornire una soluzione *platform-independent*.

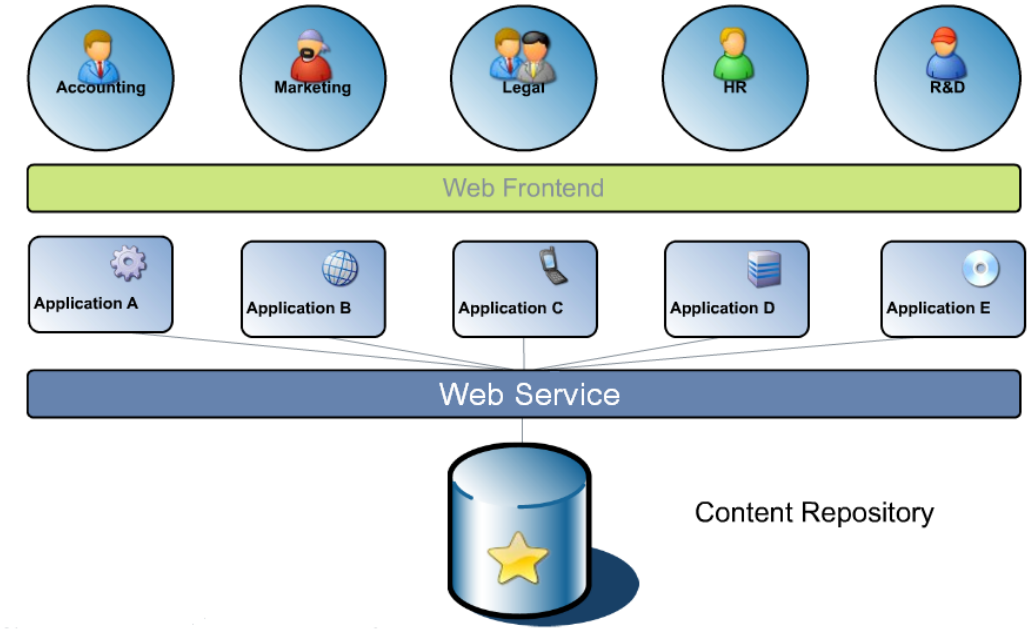


Figura 2.2: L'interoperabilità tra diverse applicazioni

Il primo passo è, quindi, quello di separare nettamente, come del resto avviene anche in modelli di programmazione basati su componenti, la logica di presentazione da quella che è la logica applicativa. I Web Service, infatti, si occupano solamente della logica applicativa, e sarà, quindi, il fruitore del servizio a presentare i dati ottenuti utilizzando il servizio con stili e grafica propri spostando così il focus del service provider al contenuto e non alla presentazione. Prendiamo in considerazione un'azienda specializzata in metereologia che voglia offrire, mediante web service, un servizio di previsioni meteo di una località a richiesta, essa, nel modello che rappresenta la logica dei servizi web, viene chiamato "Service Provider", e quindi pubblicherà il servizio con le specfiche per l'interrogazione e lo renderà visibile a chiunque voglia utilizzarlo.
Ora pensiamo ad uno sviluppatore che vuole creare un applicazione o un sito web, e voglia

offrire ai propri visitatori, o utilizzatori, il servizio personalizzato di previsioni Meteo. Lo sviluppatore, che nel modello chiameremo "Service Requestor", per implementare il servizio, utilizzerà la risorsa messa a disposizione dal "Service Provider", interrogando l'application server su cui è installato il servizio dell'azienda metereologica, e il contenuto delle risposte che otterrà, verranno utilizzate all'interno dell'applicazione sviluppata, preoccupandosi solo della rappresentazione grafica.
Alla luce di quanto descritto, si può notare come gli attori in causa siano necessariamente il fornitore (Service Provider) e il richiedente (Service Requestor). Questo tipo di paradigma, è il medesimo che si riscontra nella tipica interazione di tipo client-server che viene utilizzata nel modo tradizionale. Attraverso la SOA questa interazione viene arricchita con un ulteriore attore detto "Service Directory" o "Service Broker" che, come mostrato nella figura, si inserisce all'interno della comunicazione tra fornitore e fruitore del servizio.
Quest'ultimo attore si occupa della gestione del registry, permettendo, a chi ha necessità, di ricercare un servizio sulla base di caratteristiche con le quali è stato definito e memorizzato. Naturalmente, il Service Directory può seguire politiche di controllo degli accessi sulle interrogazioni in modo da limitare la visibilità sui servizi inseriti.

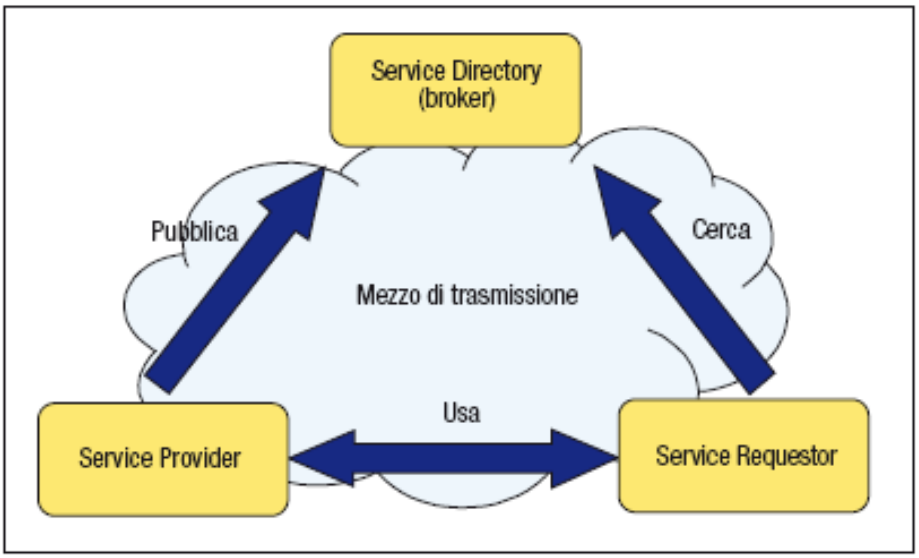


Figura 2.3: Modello SOA

Va notato che i tre attori interessati possono essere distribuiti e possono utilizzare piattaforme tecnologiche differenti, con l'unico vincolo però di dover utilizzare tutti e tre un canale trasmissivo comune. Rimanendo sul mezzo trasmissivo, questo risulta essere un parametro dell'architettura, quindi l'approccio adottato dalle SOA ha il vantaggio di potersi integrare con diversi ambienti permettendo in tal modo di realizzare applicazioni multi-canale, fruibili cioè attraverso diversi dispositivi.

L'architettura dei Web Service è stata pensata per supportare l'interoperabilità tra diversi sistemi basando la comunicazione tra i diversi soggetti su un linguaggio comune. La comunicazione avviene mediante uno scambio di informazioni descritte su un formato chiamato eXtensible Markup Language (XML).

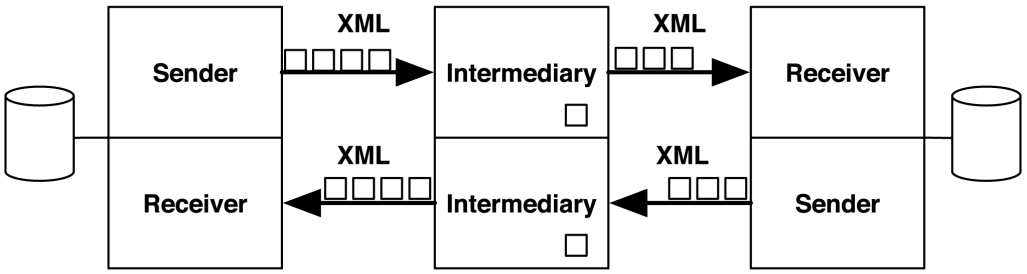


Figura 2.4: Comunicazione basata su Xml

Xml è stato sviluppato dalla W3C e consiste in un linguaggio di markup che consente di avere uno standard flessibile aggiungendo una tipizzazione sui dati. Questo formato aiuta ad esprimere concetti e informazioni in maniera standard e autodescrittiva in maniera tale che qualsiasi piattaforma, possa leggere ed elaborarne il contenuto indipendentemente dalla tecnologia da cui è stato creato.
Questa autodescrizione viene effettuata grazie all'XML Schema che è, a sua volta, in formato Xml e descrive quali sono le informazioni contenute e la struttura che il documento deve avere per essere elaborato correttamente da altri sistemi.
Esistono molte implementazioni tecnologiche che rispecchiano i concetti dell'architettura orientata ai servizi, ma non tutte riescono ad esprimere il concetto fondamentale di implementare entità che descrivano se stesse,tranne la tecnologia che sfrutta il protocollo chiamato SOAP (Simple Object Access Protocol) e definito per eccezione come il "Web Service". Quando ci si riferisce ad un Web service, si fa subito riferimento al protocollo SOAP e alla potenza del suo linguaggio di definizione dei servizi chiamato WSDL (definito nel prossimo capitolo), ma oggi, grazie alla diffusione del Web 2.0, quando si parla di Servizio Web si intende un sistema software progettato per supportare l'interoperabilità tra sistemi e per erogare contenuti in quanto, per ragioni di comodità e altri fattori che analizzeremo, oltre al classico SOAP, si è diffuso l'utilizzo di un'altra architettura denominata REST (REpresentational State Transfer), ed è su di essa che si concentra il lavoro descritto in questo documento. Nel documento verranno usati i termini "Servizio REST" e "Servizio SOAP" per differenziare la diversa tecnologia sul quale un servizio Web è implementato.

2.1.1 SOAP (Simple Object Access Protocol)

Nonostante uno dei primi scopi di SOAP sia stato quello di supportare l'RPC (Remote Procedure Call) [18] sul Web questo protocollo fornisce un meccanismo semplice e leggero per lo scambio di informazioni strutturate e tipizzate tra peers in un ambiente decentralizzato e distribuito tramite XML.
La parola ***Object*** contenuta nell'acronimo, manifesta che l'uso del protocollo dovrebbe effettuarsi secondo il paradigma della programmazione orientata agli oggetti, infatti SOAP descrive un meccanismo semplice per esprimere la semantica dell'applicazioni fornendo dei meccanismi per la codifica dei dati all'interno dell'entità coinvolte. SOAP è un framework che può operare sopra varie pile protocollari e le chiamate alle procedure remote possono essere modellizzati come uno scambio di messaggi SOAP.

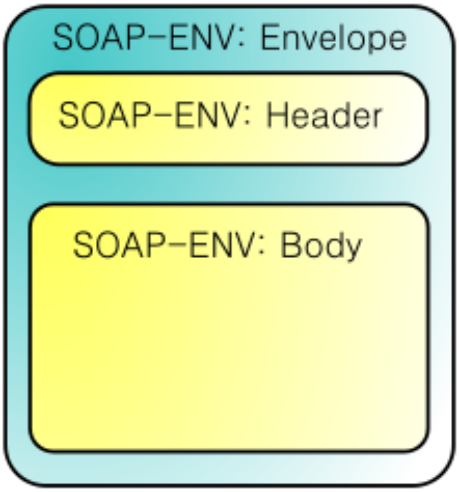


Figura 2.5: Struttura di un messaggio SOAP

La struttura di un messaggio SOAP è costituita da tre parti principali:

- **"SOAP Envelope"** identifica il documento come un messaggio SOAP, esprime ciò che si trova nel messaggio e che cosa tratta.
- **"SOAP Header"** è un elemento opzionale ma permette di definire meccanismi di serializzazione che sono usati per codificare i tipi di dati che sono istanziati all'interno del messaggio. Viene anche utilizzato per definire messaggi con diversi destinatari nel caso il messaggio dovesse attraversare più punti di arrivo.
- **"SOAP Body"** è un elemento indispensabile che contiene le informazioni scambiate dalle richieste/risposte dell'invocazione di un servizio. Il contenuto di questo elemento è definito dall'applicazione e non dalle specifiche del SOAP, anche se vengono comunque definite regole a proposito di come tali elementi devono essere manipolati.

Prendiamo ad esempio lo scenario delle previsioni meteo descritto nel Capitolo 1. Supponiamo che per avere le previsioni metereologiche di una località, siano necessarie solo le coordinate geografiche di essa e il servizio fornisca le previsioni per la giornata attuale.
Lo sviluppatore, per interrogare il servizio, dovrebbe produrre un messaggio SOAP simile a quello riportato qui sotto.

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
        xmlns:xsd="http://previsioni.example.com/xsd"
        soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <soapenv:Header/>
   <soapenv:Body>
      <xsd:Previsioni>
         <xsd:latitudine>62.33</xsd:latitudine>
         <xsd:longitudine>31.22</xsd:longitudine>
      </xsd:Previsioni>
   </soapenv:Body>
</soapenv:Envelope>
```

Il messaggio creato, viene spedito su un canale di trasmissione che permette di raggiungere l'applicazione da interrogare e la risposta che otterremo sarà simile a:

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
        soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <soapenv:Body>
      <ns:RisultatoPrevisioni xmlns:ns="hhttp://previsioni.example.com/xsd">
         <ns:Tempo>nuvoloso</ns:Tempo>
         <ns:Temperatura>+2</ns:Temperatura>
      </ns:RisultatoPrevisioni>
   </soapenv:Body>
</soapenv:Envelope>
```

Come si nota nei due messaggi scambiati, il contenuto dell'elemento "soapenv:Body" non è descritto nelle specifiche del protocollo SOAP, ma è definito dall'applicazione.
Abbiamo detto che la busta SOAP viaggia su un canale di trasmissione, ma non abbiamo definito nulla su di esso; questa è un'ulteriore caratteristica dei Web Service, infatti tutti i concetti finora espressi, rimangono ancora validi anche se il protocollo di trasmissione utilizzato per invocare la procedura è diverso. Principalmente, i canali trasmissivi maggiormente usati, sono gli stessi diffusi nel Web.

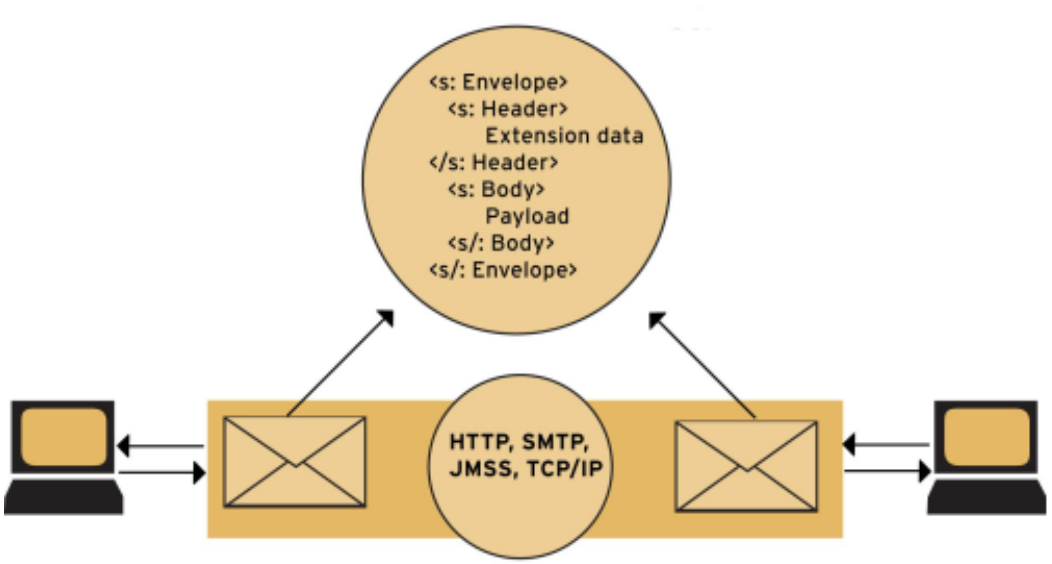


Figura 2.6: Comunicazione SOAP

Ad oggi, quello più usato nella comunicazione è il protocollo HTTP (Hyper Text Transfer Protocol) e questa ultima caratteristica pone SOAP in una posizione privilegiata rispetto ad altri meccanismi di invocazione presenti in standard di computazione distribuita quale COM+, Java RMI e CORBA, perchè quest'ultimi mostrano dei limiti nell'uso del Web durante la comunicazione.

2.1.2 REST (REpresentational State Transfer)

REST è l'acronimo di Representational Transfer State, è stato coniato nel 2000 nella tesi di dottorato "Architectural Styles and the Design of Network-based Software Architectures" [15] di Roy Fielding, uno dei principali autori delle specifiche dell' Hypertext Transfer Protocol (HTTP).
REST non è ne una tecnologia ne un'architettura, ma è un insieme di principi e metodi di progettazione (design pattern) che i sistemi ipermediali distribuiti dovrebbero avere per soddisfare determinate caratteristiche come, ad esempio, essere altamente scalabili.
Fielding la definisce come:

> *Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.*

I concetti di **architettura software** (astrazione degli elementi di run-time di un sistema, durante una delle sue fasi operative) e **stile** (insieme di vincoli che inducono le qualità

e i compromessi di un'architettura) che caratterizzano il design pattern denominato REST sono:

- **stateless** (la qualità di migliorare la scalabilità, l'affidabilità e la visibilità)
- **client-server** (separa le competenze e migliora la scalabilità)
- **caching** (migliora l'efficienza e la scalabilità)
- **interfaccia uniforme** (migliora la visibilità, semplifica e disaccoppia)
- **codice on-demand** (download ed esecuzione di applet e script: semplifica i client, aumenta le possibilità di estensione)

Bisogna affrontare un piccolo approfondimento sul concetto di "stateless", questo vincolo deriva direttamente dalla parte di acronimo ST che sta per *State Transfer*, infatti ad ogni interazione, devono essere trasmesse tutte le informazioni necessarie al server per elaborare la richiesta senza controllare l'history della comunicazione.
Questo atteggiamento compromette le performance portando ad un degrado delle prestazioni a causa del maggior numero di dati da presentare, ma permette di introdurre la terza caratteristica elencata, il "caching", quindi di poter sfruttare tecnologie proxy tra il canale trasmissivo migliorando la scalabilità, facendo attenzione ad attuare dei meccanismi robusti nel controllo delle informazioni memorizzate dato che possono essere non aggiornate.

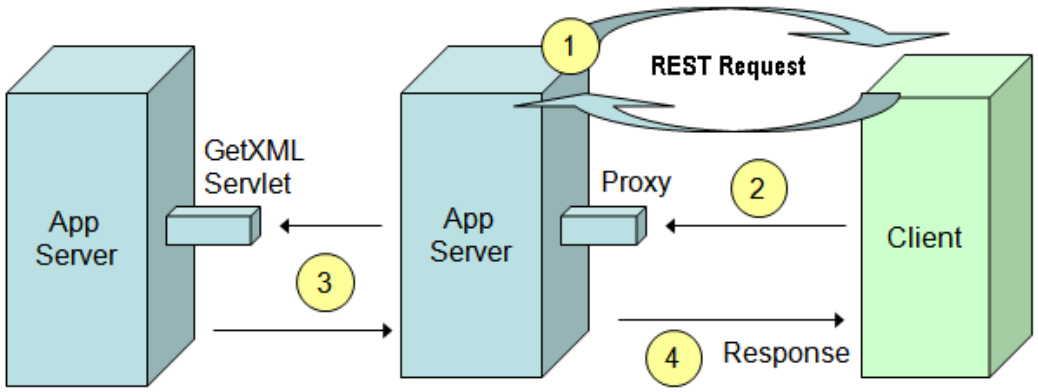


Figura 2.7: Esempio di architettura Proxy con REST

I web service costruiti sui questi concetti, vengono chiamati anche *RESTful* web service.
Il principio fondamentale su cui ruota un servizio REST, è il concetto di "Risorsa" (Resource). Nella letteratura, questi servizi vengono definiti, come implementazione di architetture ROA, ovvero "Architettura Orientata alla Risorsa".

Risorsa (Resource)

La Risorsa è l'elemento chiave di un design RESTful, in opposizione ai "metodi" e "servizi" rispettivamente usati nell'RPC e nelle richieste SOAP dei servizi WEB.
Una Risorsa è tutto ciò che è abbastanza importante da poter essere descritta in maniera autonoma. Se gli utenti possono "creare un collegamento ipertestuale su di essa, recuperare o memorizzare una rappresentazione di essa, poter includere una parte o tutte le informazioni in un'altra rappresentazione, o eseguire altre operazioni su di esso", allora questa deve essere creata come risorsa.
Questo pensiero è simile alla programmazione ad oggetti, solo che l'approccio è fondamentalmente diverso. Prendiamo ad esempio un sistema informativo che gestisce la lista di passeggeri sui voli di una compagnia aerea, e vogliamo costruire un servizio web che permetta di gestire le liste effettuando le solite operazioni di inserimento e aggiornamento su di esse. In questo scenario, un sistema informativo che propone delle soluzioni minimali, si possono individuare due risorse: il passeggero e la lista.
Per descrivere una risorsa, bisogna individuare tre elementi che in letteratura si identificano con un triangolo, che andremo ad analizzare uno per uno.

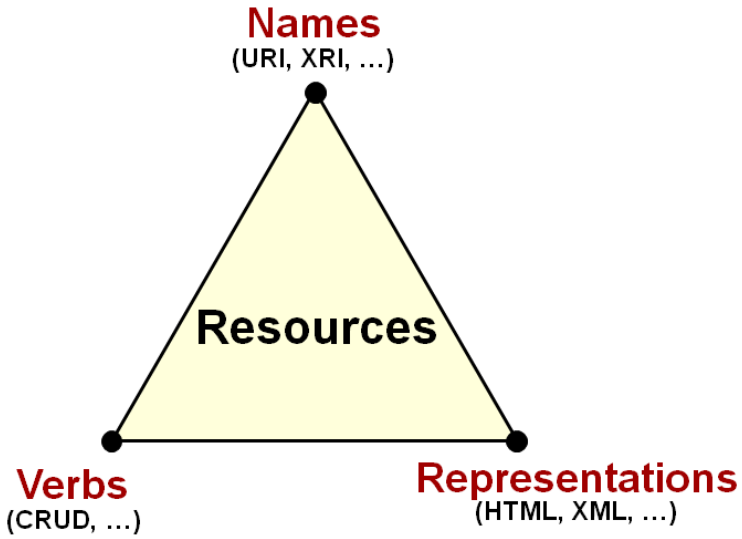


Figura 2.8: Composizione di una risorsa

Nome

Ogni risorsa deve essere identificata con un "nome", e in più deve essere univoca. Questo permette di poter identificare la risorsa in maniera non ambigua e di raggiungerla da qualsiasi altro servizio.
Il Nome può essere visto come l'indirizzo della risorsa e quindi la possibilità di dare alla risorsa la proprietà di essere indirizzabile, questa all'interno del servizio REST, viene definita mediante l'URI (Uniform Resource Identifier).
Un client che usufruisce del servizio REST, manipola la risorsa connettendosi al server che la ospita e ne invia il percorso per raggiungerla. Riprendendo l'esempio della compagnia aerea, l'URI *"www.rest-airlines.com/customers/4543892"* sta a identificare che viene richiesto al server *www.rest-airlines.com* una risorsa identificata con il percorso */costumers/4543892*.
I principi che ispirano gli URI sono ben descritte da Tim Berners-Lee in Universal Resource Identifiers-Axioms of Web Architecture [4].

> "The URI is the fundamental technology of the Web. There were hypertext systems before HTML, and Internet protocols before HTTP, but they didn't talk to each other. The URI interconnected all these Internet protocols into a Web, the way TCP/IP interconnected networks like Usenet, Bitnet, and CompuServe into a single Internet. Then the Web co-opted those other protocols and killed them off, just like the Internet did with private networks. Today we surf the Web (not Gopher), download files from the Web (not FTP sites), search publications from the Web (not WAIS), and have conversations on the Web (not Usenet newsgroups). Version control systems like Subversion and arch work over the Web, as opposed to the custom CVS protocol. Even email is slowly moving onto the Web.
> The web kills off other protocols because it has something most protocols lack: a simple way of labeling every available item. Every resource on the Web has at least one URI. You can stick a URI on a billboard. People can see that billboard, type that URI into their web browsers, and go right to the resource you wanted to show them. It may seem strange, but this everyday interaction was impossible before URIs were invented."

Nella costruzione dell'URI, esistono diverse best-practice da seguire, una su tutte è quella che la URI deve essere descrittiva, e deve essere ben strutturata. Ritornando al nostro esempio dei passeggeri, se il servizio, per conoscere i dettagli di due passeggeri, ci imponesse di costruire due URI diversi tra loro come "www.rest-airlines.com/info-about/4543892" e per il secondo "www.rest-airlines.com/get-information-on-customer/4546983" il client dovrebbe avere ulteriori informazioni che lo aiutino a distinguire quale delle due

strutture utilizzare su un cliente, se invece avesse una struttura unica, come ad esempio "www.rest-airlines.com/customers/4546983" il client può creare la sua richiesta in maniera intuitiva.

Azioni (Verbs)

Abbiamo descritto la risorsa con un nome univoco, ma la cosa più importante che un servizio web mette a disposizione, sono le operazioni che consentono di manipolare le informazioni.
Oltre a digitare gli URI per accedere alle risorse, è possibile interagire con esse per mezzo di azioni che vengono messe a disposizione del protocollo di comunicazione. A differenza della semplice consultazione, quando si eseguono questi metodi, la nostra operazione, modifica lo stato del server, ad esempio aggiungendo la nostra risorsa ad un determinato servizio. Il protocollo di trasmissione che viene utilizzato dai servizi REST è il protocollo standard HTTP [44] . I metodi utilizzati sono quelli minimi messi a disposizione GET, POST, PUT e DELETE, mentre le altre operazioni disponibili dal protocollo possono essere reindirizzate a queste quattro. Queste operazioni hanno analogia con le quattro operazioni base utilizzate nei sistemi di storage persistenti identificate con l'acronimo CRUD (Create, Retrieve, Update e Delete).

HTTP	CRUD	SQL
POST	Create	INSERT
GET	Retrive	SELECT
PUT	Update	UPDATE
DELETE	Delete	DELETE

Figura 2.9: Analogia tra Verbs e SQL

Prendiamo il nostro esempio della rest-airlines, e vediamo come eseguendo sullo stesso URI, e quindi sulla stessa risorsa, i diversi metodi, quali sono gli effetti ottenuti.

http://www.rest-airlines.com/customer

POST *-Create-*	GET *-Retrive-*	PUT *-Update-*	DELETE *-Delete-*
Creo un nuovo passeggero	Ricevo l'elenco dei clienti della compagnia		

Figura 2.10: Risultato esecuzione

Se effettuiamo una richesta con il metodo GET del protocollo HTTP, richiediamo al web service di fornirci una lista dei passeggeri che abbiano volato con questa compagnia. Se sullo stesso URI, il quale identifica la stessa risorsa, effettuiamo una POST con all'interno della richiesta, le informazioni necessarie a descrivere una nuova istanza, intendiamo richiedere al servizio di creare una nuova risorsa che identifica un nuovo passeggero.
Non tutti i Verbi possono avere effetti sulla risorsa, infatti in questo caso se effettuiamo delle interrogazioni con i due metodi mancanti, non viene eseguita nessuna risposta e in base a come verrà progettato il web service, verrà notificato una segnalazione di errore.

http://www.rest-airlines.com/customer/485347

POST *-Create-*	GET *-Retrive-*	PUT *-Update-*	DELETE *-Delete-*
	Ricevo i dettagli del passeggero corrispondente	Aggiorno i dati del passeggero	Elimino il passeggero

Figura 2.11: Risultato esecuzione

Ora proviamo ad interrogare il servizio sulla risorsa che identifica il singolo passeggero, e se accompagnamo la richiesta con il metodo GET, riceveremo le informazioni correlate al singolo passeggero.
Se utilizziamo il verbo PUT, con le informazioni necessarie, si andrà ad aggiornare i dettagli con le informazioni accompagnate dalla richiesta.
Il metodo DELETE, invocato sulla URI, richiederà al servizio di eliminare l'istanza della risorsa correlata alla URI.
I verbi POST e PUT oltre all'URI della risorsa, richiedono ulteriori informazioni per completare la richiesta, come ad esempio per il metodo PUT, bisogna fornire quali sono i dettagli da sostituire e con quali informazioni. Per fare questo, si aggiunge all'URI un frammento di codice, il quale viene denominata query, oppure all'interno del corpo della

richiesta si possono inserire le informazioni in formati che il provider richiede.
Riassumendo, nella tabella seguente, descriviamo i significati di ciascuna parte di cui si compone un URI, prendiamo in esempio:
HTTP://WWW.DOMAIN.COM/NOME/DELLA/RISORSA?NOME=JANE&COGNOME=FONDA

Parte	Esempio	Significato
schema	HTTP	Lo schema rappresenta il protocollo usato per la comunicazione con le risorse. Viene usato spesso *http* o *https* per le risorse web, ma non ci sono limitazioni allo schema che può essere usato. Esistono molti schemi standard definti [28] e un numero non precisato di schemi proprietari.
host	WWW.DOMAIN.COM	L'*host* determina il server su cui la risorsa è localizzata. Questa parte può essere opzionale in quanto l'URI contiene solamente informazioni relative che possono essere interpretate solo a livello di contesto.
path	/NOME/DELLA /RISORSA	Il *path* determina precisamente la risorsa sul server.
query	NOME=JANE& COGNOME=FONDA	La query permette di inviare informazioni addizionali al server che gli permettono di elaborare la richiesta correttamente o di modificarne la risposta. Questi valori, a volte, vengono messi nel corpo della richiesta.

Tabella 2.1: Composizione dell'URI

Rappresentazione

Abbiamo visto che il servizio RESTful si basa sul concetto di risorsa, che non sono dati veri e propri, ma solo l'idea del progettista del servizio, di come suddividere i dati in qualcosa di astratto. Ma la risposta del servizio, non può essere un'idea, deve essere una serie di informazioni strutturate in un formato specifico e con una codifica specifica.
Oltre al "Nome" e al "Verbo" la risorsa ha bisogno di un terzo elemento caratteristico, la "Rappresentazione" il quale chiude il triangolo che avevamo mostrato precedentemente.
La "Rappresentazione" è il linguaggio e la struttura con cui il servizio mostra al destinatario lo stato della risorsa. La maggior parte delle volte, le risorse stesse sono i dati e quindi una "Rappresentazione" evidenzia la risorsa stessa.
Un esempio di "Rappresentazione" di una risorsa, l'abbiamo tutt'oggi visitando alcuni siti giornalistici che mostrano le notizie in formati "printer-friendly" o visualizzano l'elenco delle notizie come documento XML, queste sono diverse "Rappresentazioni" della stessa risorsa *notizia*.
Quando si crea una *rappresentazione* di una risorsa, non si è obbligati a descrivere tutti i

dati della risorsa, ma bisogna rappresentare qualsiasi informazione di essa in maniera tale da mostrarne completamente lo stato. La rappresentazione può essere destinata all'interazione umana (ad esempio una pagina HTML) o all'interazione machine-to-machine (ad esempio XML o formati binari), infatti può essere usata anche per richiedere al servizio, di creare una nuova risorsa o di aggiornare quella esistente, inviando una rappresentazione di essa.

REST by Example

In questa sezione vogliamo mostrare un esempio completo di interazione tra un utente e un servizio Web che implementa i design pattern REST.
Ipotizziamo che un utente voglia visualizzare e aggiornare un ordine di acquisto effettuato presso l'azienda ACME. Tutte queste operazioni verranno fatte mediante un Web service REST. Nell'esempio riportato nella figura 2.12, l'utente effettua la prima invocazione chiedendo al servizio i dettagli del suo account con una semplice richiesta *http* alla risorsa *customers* identificato dal proprio nome che sarà *coyote*.

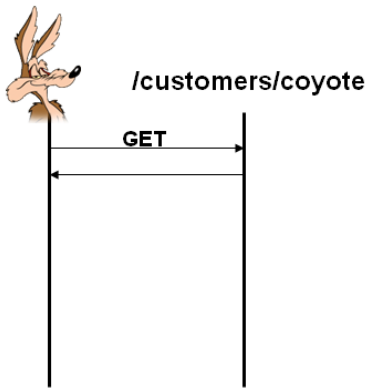


Figura 2.12: Richiesta Dettagli dell'utente

La risposta che otterrà, riportata nel Listing 2.1 sarà il dettaglio della risorsa richiesta in una "Rappresentazione" XML, all'interno della risposta si troverà il percorso della risorsa degli ordini (*orders*) da richiedere al servizio.

Listing 2.1: Dettagli dell'utente

```
<customer>
        <name>Wile E. Coyote</name>
        <portrait>http://coyote.com/my_portrait.png</portrait>
        <orders>http://acme.com/customers/coyote/orders</orders>
</customer>
```

Verrà richiesta al service i dettagli degli ordini che l'utente ha fatto tramite l'azienda ACME utilizzando l'informazione contenuta nella risposta precedente, effettuando una

normale GET. *http* alla risorsa *customers* identificato dal proprio nome *coyote*, come in figura 2.14.

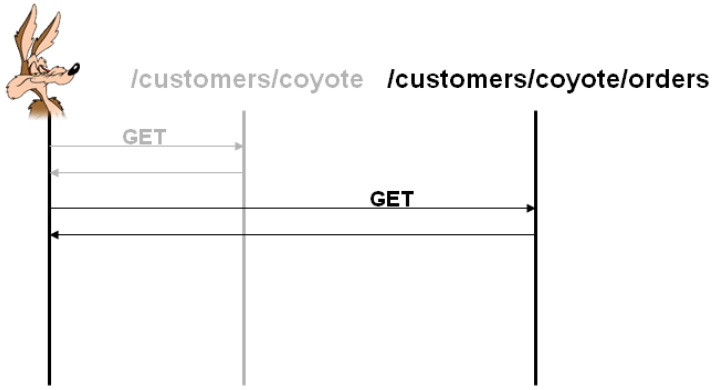


Figura 2.13: Richiesta elenco degli ordini

Nel listing 2.2 notiamo la risposta ottenuta, la quale contiene oltre all'informazione che ci aspettavamo, anche delle altre informazioni che sono presenti all'interno della risorsa customers che avevamo ottenuto nella richiesta precedente. Questo esempio mostra come le risorse, non sono entità a sè stanti, ma possono essere collegate tra di loro e nella loro rappresentazione si possono incontrare diverse informazioni correlate.
Dalla risposta otteniamo l'URI da invocare per richiedere gli ulteriori ordini effettuati se nell'elenco non esiste quello richiesto, e per ogni singolo ordine, otteniamo l'URI per poter effettuare le operazioni di tipo CRUD sul singolo ordine.

Listing 2.2: Elenco degli ordini

```
<orders>
        <customer>http://acme.com/customers/coyote</customer>
        <next>http://acme.com/customers/coyote/orders?before=11</next>
        <order id=''20''>
                <uri>http://acme.com/orders/1234</uri>
                <status>open</status>
        </order>
        ...
</orders>
```

L'ordine di cui vogliamo avere il dettaglio è identificato con l'id numero 20 e l'URI della risorsa associato all'ordine è:

http://acme.com/orders/1234.

Richiediamo il dettaglio dell'ordine, effettuando una richiesta GET all'URI

http://acme.com/orders/1234.

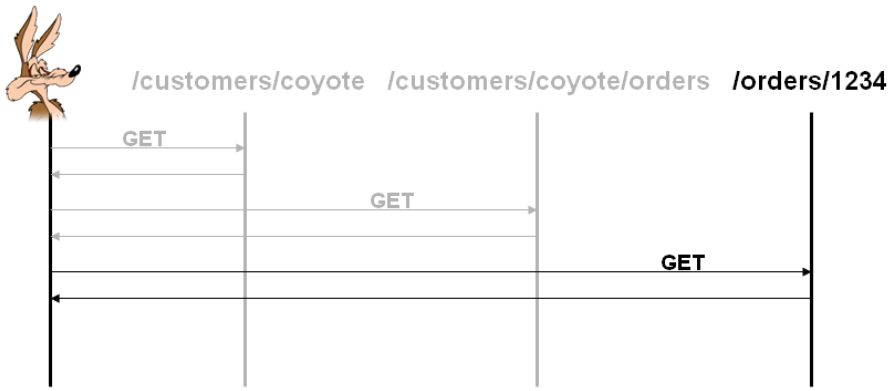


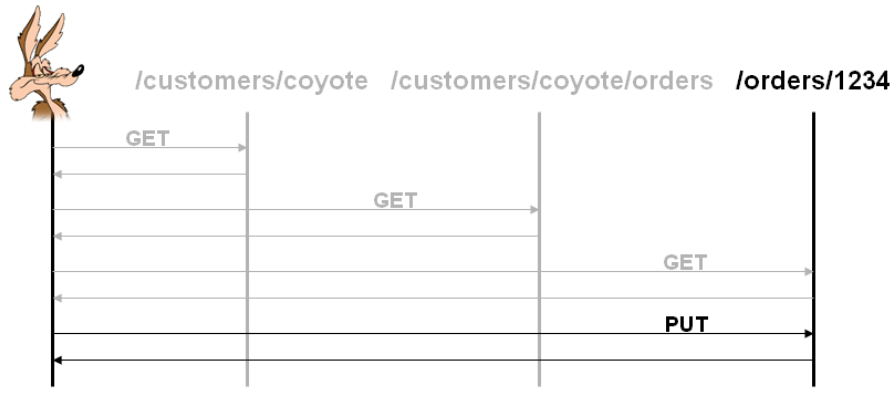
Figura 2.14: Richiesta Dettagli dell'ordine

Otteniamo la rappresentazione, Listing 2.3 della risorsa associata all'ordine e notiamo che la quantità acquistata è pari a 1.

Listing 2.3: Dettaglio dell'ordine

```
<order>
        <customer>http://acme.com/customers/coyote</customer>
        <status>open</status>
        <item quantity="1">ACME Rocket</item>
</order>
```

Ora vogliamo modificare l'ordine e aumentare la quantità di prodotti da 1 a 4. Per fare ciò, effettuiamo una richiesta allo stesso URI a cui abbiamo richiesto i dettagli dell'ordine, ma utilizziamo il metodo PUT al posto del GET e inviamo al server la rappresentazione ottenuta, modificandola con i nostri nuovi valori, come mostrato nella figura 2.15.




```
<order>
        <customer>http://acme.com/customers/coyote</customer>
        <status>open</status>
        <item quantity="4">ACME Rocket</item>
</order>
```

Figura 2.15: Modifica dell'ordine

Se la richiesta è andata a buon fine, abbiamo modificato l'ordine.
Abbiamo effettuato quattro diverse richieste per riuscire a modificare l'ordine, ma se volessimo modificarne un altro, abbiamo lo schema dell'URI che identifica la risorsa "ordine" e possiamo direttamente effettuare le ultime due operazioni senza farne delle altre.
Si noti come il servizio REST è caratterizzato dalla

- *Semplicità* in quanto è centrato sulle risorse
- *Scalabile*, in quanto abbiamo un protocollo stateless
- *Stratificabile*, in quanto i risultati delle risposte possono essere memorizzati tra gli intermediari coinvolti nella comunicazione

URI-Template

Abbiamo a disposizione molti strumenti standard o proprietari, per descrivere come viene strutturata una rappresentazione in maniera tale da poter effettuare l'interazione anche tra macchina-macchina. Ma dato che un servizio REST si basa sul concetto di risorsa e il modo con cui è possibile effettuare operazioni su di essa, è quella di sapere come costruire l'URI per potervi accedere, abbiamo bisogno di un linguaggio che ci permetta di descrivere completamente la struttura dell'URI, per questo scopo ci viene in aiuto l'URI-Template [21].
L'URI-Template ci permette di definire una serie di URI strutturalmente simili. I modelli sono composti da due parti: un percorso e una query. Il percorso è costituito da una serie di segmenti delimitati da una barra (/) e ogni segmento può avere un valore letterale, un valore variabile che deve essere visualizzato alla fine del percorso.L'espressione di query può essere interamente omessa; se presente, specifica una serie non ordinata di coppie nome/valore.
La sintassi dell'URI-Template è stata accuratamente progettata per bilanciare la necessità di un potente meccanismo di sostituzione, con facilità di implementazione e di sicurezza. La sintassi è facile da analizzare e allo stesso tempo fornisce una flessibilità sufficiente per esprimere molti comuni scenari di templating.
L'URI descritto con questo linguaggio contiene delle espressioni compatte (combinazioni di operazioni e variabili) che al momento di processare il template, vengono generate delle espressioni espanse che permettono di ricostruire l'URI.

Variabile

{nome[=<valore di default>]}

Sono identificatori letterali racchiusi tra parentesi graffe ({}). Nell'elaborazione, se viene passata una coppia nome/valore corrispondente all'identificatore, viene sostituito il valore

corrispondente. Se non viene fornito nessun valore, viene sostituito con una stringa vuota. Prendiamo in considerazione di avere piu servizi sullo stesso host, e che ogni servizio fornisca operazioni per gestire articoli di un negozio. Per avere la lista di tutti gli articoli del negozio "decathlon" bisogna effettuare una richiesta GET all'URI:

http://www.esempio.it/decathlon/articoli

Se invece vogliamo richiedere la lista del negozio "cisalfa" bisogna effettuare una richiesta GET all'URI:

http://www.esempio.it/cisalfa/articoli

Come si nota, la struttura è molto simile e la risorsa articoli la si raggiunge modificando il nome del negozio. Questa struttura si può definire come http://www.esempio.it/{nomeNegozio}/articoli e fornendo la coppia nomeNegozio=cisalfa si ottiene l'URI del secondo esempio, se invece non viene fornita nessuna coppia, si otterrebbe una URI non corretta:

http://www.esempio.it//articoli

Per ovviare a questo problema, si può dichiarare anche un valore di default alla variabile in modo tale che se non esiste nessuna coppia corrispondente alla variabile, venga sostituito con il valore dichiarato.
Nel nostro esempio, nel caso in cui non venga dichiarata nessuna coppia, vogliamo che punti al negozio decathlon e questo si definisce modificando l'URI-Template in:

http://www.esempio.it/{nomeNegozio=decathlon}/articoli

Operazione JOIN

{-join|<carattere di separazione>|VARIABILE[,VARIABILE]}*

Questa operazione permette di elencare una serie di parametri che se vengono forniti durante l'elaborazione, vengono concatenati con il carattere indicato nel template in forma coppia nome/valore. Questa operazione viene utilizzata spesso per dichiarare la struttura della query che permette di fornire delle informazioni ulteriori alla richiesta.
Il singolo parametro contenuto in questa struttura, ha le stesse caratteristiche della "Variabile", infatti anche qui, se un parametro nella query deve essere sempre presente, si può definire un valore di default che verrà sempre accoppiato al nome del parametro nel caso in cui non verrà passato al compilatore un valore corretto.

Esempio: *http://www.esempio.it/risorsa?{-join|&|foo,bar=5}*

La tabella seguente elenca il risultato della elaborazione dell'URI-Template sopra indicato, in base ai parametri che vengono forniti al compilatore.

Parametri	**Risultato**
foo=3,param=6	http://www.esempio.it/risorsa?foo=3&bar=5
bar=2,param=6	http://www.esempio.it/risorsa?bar=2
foo=3,bar=2	http://www.esempio.it/risorsa?foo=3&bar=2

Tabella 2.2: Esempio dell'operazione join

Operazione OPTION

{-opt|<Valore>|<Parametro>}

Questa operazione permette di far visualizzare un valore solo se il parametro associato viene inizializzato. É un operazione che gestisce un flag all'interno dell'URI, e se il parametro associato a questo flag viene passato all'elaboratore, viene espansa la regola con il valore che viene definito nel modello.

Esempio: *http://www.esempio.it/risorsa/{-opt|lista|param}*

Parametri	**Risultato**
param=6	http://www.esempio.it/risorsa/lista
bar=2	http://www.esempio.it/risorsa/

Tabella 2.3: Esempio dell'operazione option

Operazione NEGATIVE

{-neg|<Valore>|<Parametro>}

L'operazione Negative, ha la logica riflessa della operazione Option descritta sopra. Se viene attivato il flag, la regola non viene espansa con il valore dichiarato nel modello.

Esempio: *http://www.esempio.it/risorsa/{-neg|valore|flag}*

Parametri	**Risultato**
flag=6	http://www.esempio.it/risorsa/
bar=2	http://www.esempio.it/risorsa/valore

Tabella 2.4: Esempio dell'operazione negative

Operazione LIST

{-list|<carattere>|<parametro>}

Abbiamo sempre parlato e mostrato esempi di parametri che usano valori singoli, ma nella programmazione moderna, si può associare ad una variabile, un elenco di valori, questi vengono detti *array*. Anche questo linguaggio di modellazione deve supportare operazioni su un elenco di valori. L'operazione *List* permette di impostare un parametro e di associargli un elenco di valori, il risultato dell'elaborazione sarà la lista concatentata dal carattere che viene indicato nella modellazione. Permette di descrivere delle URI con strutture complesse.

Esempio: *http://www.esempio.it/risorsa/{-list|/|foo}*

Parametri	Risultato
foo=[fred, barney, wilma]	http://www.esempio.it/risorsa/fred/barney/wilma
foo=[a, “, c]	http://www.esempio.it/risorsa/a//c
foo=[betty]	http://www.esempio.it/risorsa/betty
foo=[]	http://www.esempio.it/risorsa/
bar=[betty]	http://www.esempio.it/risorsa/

Tabella 2.5: Esempio dell'operazione list

Operazione SUFFIX

{-suffix|<carattere>|<parametro>}

Anche questa struttura permette di elaborare liste di valori. Viene elaborata con la stessa logica dell'operazione *list* con l'eccezione che questa operazione pospone ad ogni valore dell'elenco associato al parametro, il carattere inserito nel descrittore del modello.

Esempio: *http://www.esempio.it/risorsa/{-suffix|/|foo}*

Parametri	Risultato
foo=[fred, barney, wilma]	http://www.esempio.it/risorsa/fred/barney/wilma/
foo=[a, “, c]	http://www.esempio.it/risorsa/a//c/
foo=[betty]	http://www.esempio.it/risorsa/betty/
foo=[]	http://www.esempio.it/risorsa/
bar=[betty]	http://www.esempio.it/risorsa/

Tabella 2.6: Esempio dell'operazione suffix

Operazione PREFIX

{-prefix|<carattere>|<parametro>}

Come si intuisce dal nome, questa operazione antecede, ad ogni valore dell'elenco, il carattere indicato. É l'operazione riflessa del comando precedente.

Esempio: *http://www.esempio.it{-prefix|/|foo}/dopo/la/regola*

Parametri	Risultato
foo=[fred, barney, wilma]	http://www.esempio.it/fred/barney/wilma/dopo/la/regola
foo=[a, “, c]	http://www.esempio.it/a//c/dopo/la/regola
foo=[betty]	http://www.esempio.it/betty/dopo/la/regola
foo=[]	http://www.esempio.it/dopo/la/regola
bar=[betty]	http://www.esempio.it/dopo/la/regola

Tabella 2.7: Esempio dell'operazione prefix

2.1.3 REST vs SOAP

In questa sezione vogliamo analizzare le differenze che le due tipologie di servizi hanno, per poter identificare meglio i vantaggi e svantaggi delle due metodologie.
Un servizio web basato sulla metodologia REST, ha come punto di forza l'utilizzo di un interfaccia che è diffusamente conosciuta: URI. Qualsiasi client e server che supporta invocazioni HTTP, può facilmente invocare un servizio REST.
In un servizio SOAP le richieste vengono descritte da un linguaggio chiamato WSDL, quindi questo comporta che per poter usare al meglio un servizio SOAP, si abbia la competenza di saper leggere un file XML che descrive tutte le informazioni necessarie per invocare le operazioni.
Avere una rigida descrizione del servizio, è una caratteristica che in alcuni scenari può essere un vantaggio, come ad esempio la comunicazione di scambio nei grandi Data Center dove le informazioni, per motivi di sicurezza devono essere tipizzate.
Un servizio SOAP, indirizza le richieste sempre verso un unico indirizzo chiamato “*endpoint*” e all'interno del messaggio vengono definite le operazioni che vengono parsate dall'application server ed invocate insieme ai dati sempre dichiarati all'interno del messaggio.

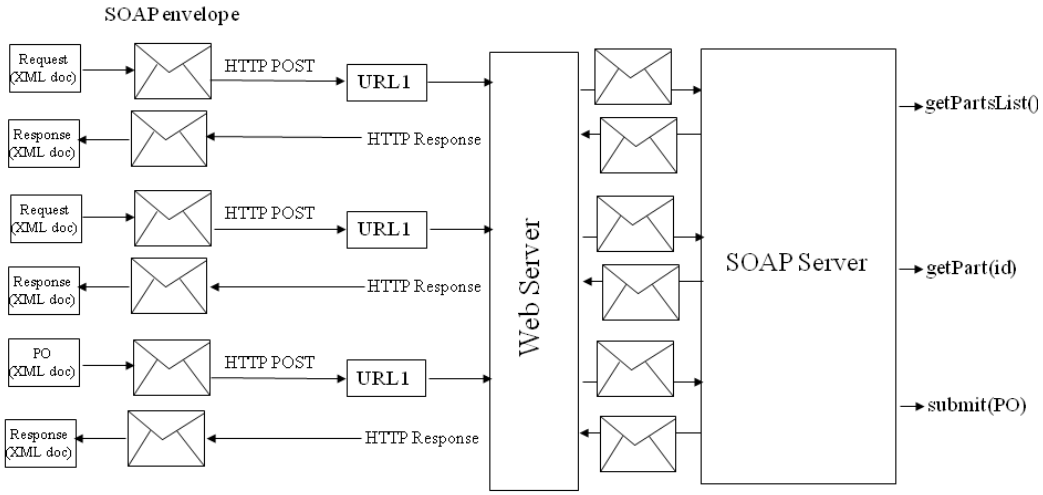


Figura 2.16: Richieste SOAP

Nella figura 2.16 si nota come tutte le richieste vengono effettuate verso un'unica URI (*URL1*). Tutte le richieste vengono effetuate sempre con il metodo POST e questo porta ad un maggior traffico verso l'end-point in quanto in tutte le richieste viene sempre inviato un documento XML.
Nei servizi REST le richieste, come detto precendentemente, vengono indirizzate verso URI differenti che si mappano sulle risorse. Il consumo di banda che porta un servizio REST è ridotto al minimo, infatti viene inviato insieme alla richiesta un documento XML o altre informazioni oltre all'URI solo quando bisogna creare o aggiornare lo stato di una risorsa.

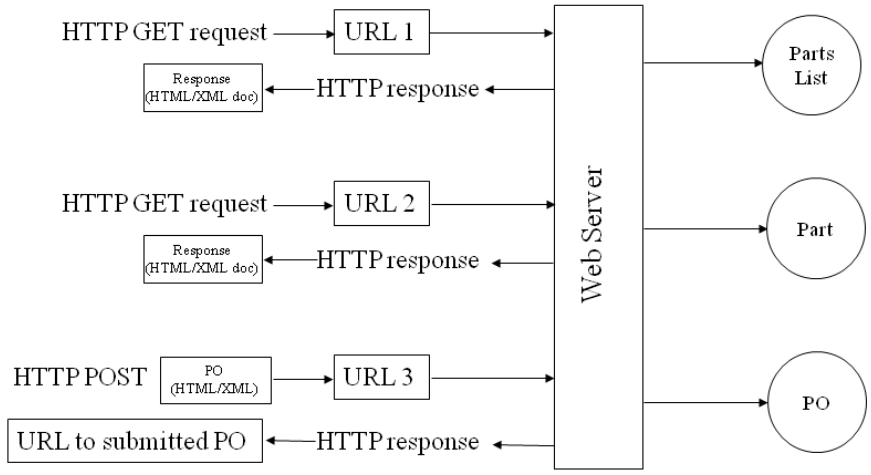


Figura 2.17: Richieste REST

La composizione dei messaggi di un servizio SOAP, sono complessi e includono infor-

mazioni che non possono essere salvate poichè non riusabili, questo non permette l'utilizzo di proxy e ne compromette la scalabilità.
L'aspetto più interessante da analizzare tra le due tipologie di servizi riguarda la sicurezza. Nella comunicazione di un servizio REST, gli apparati di scurezza come il firewall sono in grado di discernerne l'intento per ciascun messaggio, analizzando il comando HTTP utilizzato nella richiesta. Ad esempio, una richiesta GET può sempre essere considerata sicura, in quanto non può, per definizione, modificare nessun dato.
Dall'altra parte, una tipica richiesta SOAP, utilizza il metodo POST per comunicare con un servizio e gli apparati senza un'analisi completa dei messaggi, controllandone il contenuto, non si è in grado di predire se è una richiesta che può modificare le informazioni e quindi attuare gli opportuni meccanismi di controllo.
Per la parte di autenticazione e autorizzazione, SOAP utilizza una sua metodologia di autenticazione inserita nella specifica del protocollo di comunicazione demandando tutto in fase di progettazione del servizio.
La metodologia REST, invece, si basa sul fatto che i Web server supportano già questo tipo di operazioni tramite l'uso di standard come lo scambio di certificati e comuni sistemi di gestione dell'identità (come ad esempio un LDAP server).
Questo disaccoppiamento a volte può aiutare lo sviluppatore del servizio a demandare la problematica dell'autenticazione ad applicazioni esterne che sono facilmente integrabili con l'applicazione RESTful. Altre caratteristiche che differenziano i due tipi di servizio, sono la complessità che si incontra nella costruzione delle applicazioni lato client e lato server. Effettuare chiamate HTTP ad un servizio è molto semplice e non richiede l'utilizzo di librerie complesse per creare e leggere pacchetti di messaggi come avviene in un servizio SOAP.
Dal lato server, la complessità si incontra nell'esposizione delle API del servizio, in quanto la gestione del messaggio di una richiesta SOAP, avviene mediante librerie del server, mentre in un servizio REST bisogna gestire la serializzazione del file XML in uscita dalle richieste. Un punto critico è l'esposizione del servizio all'utente che dovrà utilizzarlo, infatti nel SOAP viene messa a disposizione il WSDL, che permette di serializzare in un formato standard le API del servizio. Nei servizi REST viene, ad oggi, utilizzata una pagina in HTML human-readable e quindi è impossibile automatizzare la creazione di richieste da parte di un calcolatore fornendo solo la sua descrizione.
Queste differenze fanno intuire che un servizio RESTful non permette, la sostituzione di un servizio creato su SOAP, ma ne è un'alternativa,ogni tipologia ha vantaggi e svantaggi nella sua applicazione. L'uso di REST o di SOAP è sicuramente determinato dal contesto in cui si vuole utilizzarlo.

2.2 Linguaggi Descrittivi

Abbiamo fino adesso descritto i principi e le regole delle due tipologie di servizi web che abbiamo preso in considerazione.
In questa sezione vogliamo introdurre i linguaggi che ci permettono di fornire una descrizione delle funzionalità che i servizi offrono all'utente consumatore o che permetta un'automazione di comunicazione tra servizi.

2.2.1 WSDL

I servizi che vengono sviluppati con la tecnologia e con i principi SOAP, sono servizi che vengono descritti molto rigidamente e questa descrizione viene fatta grazie a un linguaggio che si basa su XML e chiamato WSDL [13]. Un "documento" WSDL contiene, relativamente al Web Service descritto, informazioni su:

- *cosa* può essere utilizzato (le "operazioni" messe a disposizione dal servizio);
- *come* utilizzarlo (il protocollo di comunicazione da utilizzare per accedere al servizio, il formato dei messaggi accettati in input e restituiti in output dal servizio ed i dati correlati) più precisamente i "vincoli" (*bindings* in inglese) del servizio;
- *dove* utilizzare il servizio (cosiddetto *endpoint* del servizio che solitamente corrisponde all'indirizzo - in formato URI - che rende disponibile il Web Service)

Si supponga che una banca voglia fornire un servizio di approvazione automatica prestiti di piccola entità che permette ai correntisti della banca di chiedere un prestito direttamente on-line. Dando per certo che l'utente si sia precedentemente registrato, questi non dovrà far altro che indicare il proprio identificativo che corrisponderà al proprio numero di conto corrente e la somma di denaro richiesta.
Già da questa breve descrizione è possibile identificare come il servizio metta a disposizione dell'utente un'operazione (*approva*, per esempio) che richiederà dei dati in ingresso e in uscita. Attraverso WSDL è possibile formalizzare tutte queste caratteristiche del servizio secondo uno schema che non differisce molto dalla specifica delle API (*Application Program Interface*) di un sistema.

```
<definitions targetNameSpace=''http://example.it/web_services/approvaPrestito''
xmlns:tns=''http://example.it/web_services/approvaPrestito''
xmlns:xsd=''http://www.w3.org/2001/XMLSchema''
xmlns=''http://schemas.xmlsoap.org/wsdl/''>
<types>
...
</types>
<message name=''Richiedente''>
<part name=''contocorrente'' type=''xsd:integer'' />
<part name=''ammontare'' type=''xsd:integer'' />
```

```
</message>
<message name=''messaggioErrore''>
<part name=''codice'' type=''xs:integer'' />
</message>
<message name=''approvazione''>
<part name=''risposta'' type=''xs:string'' />
</message>
<portType name=''approvazionePrestitoPT''>
<operation name=''approvaPrestito''>
<input message=''tns:richiedente'' />
<output message=''tns:approvazione'' />
<fault name=''errore'' message=''tns:messaggioErrore'' />
</operation>
</portType>
<binding name=''approvaPrestitoSOAPBinding'' type=''tns:approvazionePrestitoPT''>
<soap:binding style=''document''
transport=''http://schemas.xmlsoap.org/soap/http''/>
<operation name=''approvaPrestito''>
<soap:operation soapAction=''http://www.example.it/web_services/approva''/>
<input>
<soap:body use=''literal''/>
</input>
<output>
<soap:body use=''literal''/>
</output>
</operation>
</binding>
<service name=''approvaPrestitoService''>
<documentation>
Servizio di approvazione prestiti automatico
</documentation>
<port name=''approvaPrestito'' binding=''approvaPrestitoSOAPBinding''>
<soap:address location=''http://www.example.it/web_services/approva''/>
</port>
</service>
</definitions>
```

Come si può notare dall'esempio qui sopra, il WSDL è un linguaggio basato su XML e permette di specificare quelle caratteristiche del servizio che prima sono state descritte solo a parole.
Il componente fondamentale di un file WSDL è il tag *service* che identifica un insieme logico di servizi che possono essere più di uno.
Ogni servizio specificato, viene identificato dal nome e il tag opzionale *documentation* permette di specificare una breve descrizione del servizio. All'interno del tag *service* viene specificato l'indirizzo su cui risponde il servizio mediante il tag *soap:address* il quale si trova all'interno del tag *port* e identifica, mediante l'attributo *binding* le caratteristiche del servizio.
Associato all'attributo del tag port descritto sopra, viene identificato il nome del tag *binding* che identifica il protocollo di trasporto (SOAP, HTTP o SMTP), il nome dell'operazione che deve essere specificato all'intero del messaggio SOAP e la sintassi dei messaggi

di input e output che possono essere definiti all'interno di questo tag o mediante al tag *portType*.
Il tag *portType* il cui compito è quello di specificare la reale sintassi del servizio, ha, quindi, il ruolo di dire cosa il servizio fa, mentre la port come il servizio possa essere acceduto.
Con *portType* si caratterizza la tipologia di operazione, infatti, vengono identificate quattro diverse classi di operazioni.

- *One_way*: l'operazione è composta da un solo messaggio in ingresso al fornitore del servizio.
- *Request_response*: l'operazione prevede una risposta del fornitore del servizio successiva a un messaggio ricevuto dall'utente.
- *Solicit_Response*: l'operazione prevede l'attesa da parte del fornitore del servizio, di una risposta a fronte di una richiesta effettuata dal fornitore stesso.
- *Notification*: l'operazione è composta da un solo messaggio in uscita al fornitore del servizio.

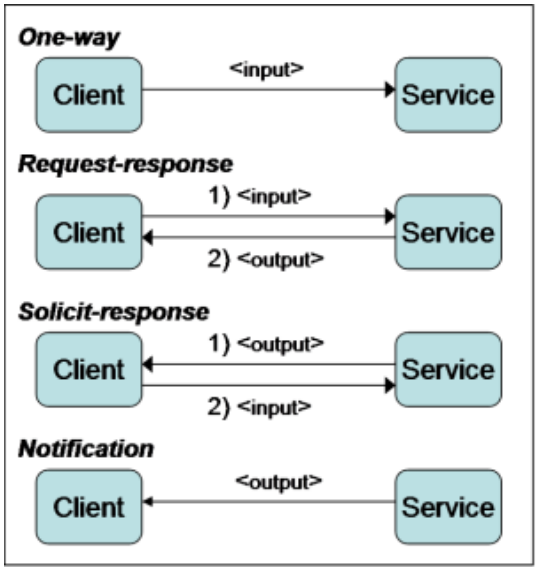


Figura 2.18: Tipologia di operazioni SOAP

Riferendosi, quindi, all'esempio, viene specificato un solo servizio (ApprovaPrestito) così come appare nel tag port. Questo servizio è accessibile all'indirizzo

http://www.example.it/web_services/approva

e per invocarlo è necessario che il client inoltri le proprie richieste secondo il protocollo SOAP.
Al di là delle specifiche sintattiche di WSDL, soffermandosi ulteriormente sulla relazione

che esiste tra *port* e *portType* è possibile dire che una *port* può essere vista come una specializzazione di una *portType* operata secondo un particolare protocollo di comunicazione.
È quindi possibile che all'interno del medesimo file WSDL la stessa *portType*, quindi il servizio, possa essere reso accessibile, grazie ai *binding*, su diversi protocolli di trasporto.

2.2.2 SAWSDL

Il linguaggio WSDL descrive la sintassi e le caratteristiche dei messaggi che vengono scambiati durante la comunicazione tra il web service e l'utente, ma non descrive la semantica dei dati che vengono scambiati e quindi la descrizione tramite WSDL viene solo interpretata dalla macchina ma non capito, rendendo molto difficile la condivisione delle informazioni.
Il nostro obiettivo è quello di definire dei meccanismi che permettono di trasformare il linguaggi WSDL da machine-representable a machine-understandable specificando le relazioni tra le informazioni scambiate e permettendo una loro elaborazione automatica, generando documenti che possano non solo essere letti ed apprezzati da esseri umani, ma anche accessibili ed interpretabili da agenti automatici.
A tale scopo sono stati definiti alcuni linguaggi, quali Resource Description Framework (RDF) e Web Ontology Language (OWL), entrambi basati su XML, che consentono di esprimere le relazioni tra le informazioni, rifacendosi alla logica dei predicati mutuata dall'intelligenza artificiale.
Una delle tecnologie più promettenti utilizzate in questo ambito è chiamata Semantic Annotations for WSDL (SAWSDL) [24]. SAWSDL nasce come evoluzione del precedente WSDL-S (Web Service Semantics), proposto al W3C dal Large Scale Ditributed Information Systems (LSDIS) dell'Università della Georgia negli Stati Uniti.
Il SAWSDL viene applicato alla descrizione basata su WSDL inserendo informazioni mediante l'utilizzo di tre attributi:

- *sawsdl:modelReference* è l'associazione tra i componenti del WSDL e Xml Schema a specifici concetti in un modello semantico.
- *sawsdl:liftingSchemaMapping* utilizzato per indicare la presenza di concetti multipli racchiusi in un unico componente delWSDL e Xml Schema e il collegamento ad un file che ha il compito di esplicitare i concetti e collegarli alla ontologia.
- *sawsdl:loweringSchemaMapping* utilizzato per effettuare l'operazione inversa, ossia racchiudere in un unico componente del WSDL o Xml Schema i diversi concetti che questo componente racchiude.

Il SAWSDL descrive il significato semantico delle informazioni e, a differenza della descrizione sintattica in cui bisogna fornire informazioni complete sulla struttura dei

messaggi e delle operazioni, per il SAWSDL bisogna fornire informazioni che permettono di distinguere i concetti dalla struttura.
Quello appena esposto è un processo solo apparentemente tecnico, ma che ha come obiettivo, l'approdo all'intelligenza condivisa del web che promette, un'autentica trasformazione nella natura del software e dei servizi.

Tra i principi di progettazione dei servizi web RESTful, non è definito nessun metodo e nessun meccanismo che consente di descrivere le caratteristiche del servizio in maniera tale da poterne elaborare con agenti automatici. Quasi tutti i servizi REST, vengono descritti ad oggi con pagine statiche HTML che non possono essere effettuate elaborazioni su di esse, ma sono solo human-readable.
L'interazione che avviene tra un client e un servizio RESTful è una serie di operazioni dove il client invia una richiesta ad una risorsa e ne riceve una risposta.

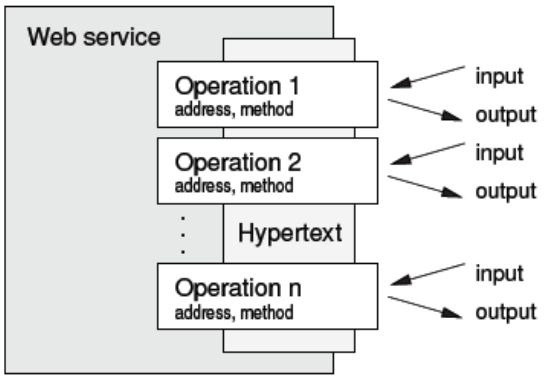


Figura 2.19: Modello funzionale di un servizio Web RESTful

Per descrivere un servizio RESTful si ha bisogno di focalizzare dei concetti chiave:

- *service*, si indica il servizio come un insieme di risorse correlate
- *operation*, è una singola azione che il client può eseguire sul servizio
- *resource*, determina gli indirizzi (URI) dove le operazioni devono essere invocate
- *method*, cattura i metodi HTTP per ogni operazione
- *request* e *response*, descrivono la composizione dei messaggi inviati come input e la risposta delle operazioni.

La descrizione di un'operazione viene specificata da un indirizzo (un URI anche in formato URI-Template), il metodo HTTP da invocare, e la struttura con i formati dei dati

in input e output. Principalmente il formato del messaggio di risposta può essere auto-descrittivo ma la documentazione delle API dovrebbe specificare cosa il client si dovrebbe aspettare per poterne elaborare automaticamente.
In questa sezione descriveremo alcuni linguaggi che vengono proposti per creare una descrizione a livello sintattico e semantico dei servizi RESTful.

2.2.3 hRESTS

Lo scopo di hRESTS [22] è quello di fornire una descrizione in formato machine-readable del Web Service e delle sue API. hRESTS è un microformat che consente di inserire delle informazioni di mark-up sul documento di descrizione in HTML del servizio.
hRESTS sfrutta le caratteristiche dell'XHTML come l'utilizzo degli attributi *class* e *rel* rendendo disponibile, il frammento di descrizione su cui si utilizzano, ad una facile elaborazione. Il Listato 2.4 mostra una realizzazione del modello RDFS di un servizio, insieme con le proprietà descritte sopra. I servizi, le loro operazioni e i messaggi possono anche avere nomi human-readable, che possono essere collegati mediante la proprietà *rdfs:label*. Notare che viene riutilizzato il vocabolario minimo del modello RDF WSMO-Lite [42].

Listing 2.4: Modello hRESTS in RDFS

```
@prefix hr: <http://www.wsmo.org/ns/hrests#> .
@prefix rdf: <http://www.w3.org/1999/02/22..rdf..syntax..ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf..schema#> .
@prefix wsl: <http://www.wsmo.org/ns/wsmo..lite#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# classi e proprietà del modello WSMO..Lite minimale
wsl:Service a rdfs:Class .
wsl:hasOperation a rdf:Property ;
rdfs:domain wsl:Service ;
rdfs:range wsl:Operation .
wsl:Operation a rdfs:Class .
wsl:hasInputMessage a rdf:Property ;
rdfs:domain wsl:Operation ;
rdfs:range wsl:Message .
wsl:hasOutputMessage a rdf:Property ;
rdfs:domain wsl:Operation ;
rdfs:range wsl:Message .
wsl:Message a rdfs:Class .

# hRESTS prporietà aggiunte
hr:hasAddress a rdf:Property ;
rdfs:domain wsl:Operation ;
rdfs:range hr:URITemplate .
hr:hasMethod a rdf:Property ;
rdfs:domain wsl:Operation ;
rdfs:range xsd:string .

# datatype per l'URI-Template
hr:URITemplate a rdfs:Datatype .
```

Per rendere migliore la leggibilità sia da parte dell'utente che da parte di elaboratori, viene affiancato l'utilizzo di RDFa che consiste nell'utilizzare una serie di attributi dell'XML esprimendo i dati RDF su qualsiasi linguaggio di markup, specialmente sull'HTML. Per capire meglio come si utilizza questo tipo di linguaggio, facciamo un esempio. Supponiamo di avere il nostro solito servizio di previsioni meteo e di volerlo descrivere mediante hRESTS su RDFa.

```
<div typeof="wsl:Service" about="#svc1"
 xmlns:hr="http://www.wsmo.org/ns/hrests#"
 xmlns:wsl="http://www.wsmo.org/ns/wsmo-lite#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<h1>
  <span property="rdfs:label">Weather Forecast</span> service API
</h1>
        This is a service of Weather Forecast
  <div rel="wsl:hasOperation">
  <span typeof="wsl:Operation" about="#op1">
         <h2>Operation <code property="rdfs:label">
               getForecastByLocation
             </code>
         </h2>
          <p>Invoked using the method
                 <span property="hs:hasMethod">POST</span> at
                 <code property="hr:hasAddress"
                   datatype="hr:URITemplate">
                   http://example.com/rest/service/
                       getForecastByLocation?{-join|&|lat,lon}
                 </code><br/>
                 <span rel="wsl:hasInputMessage">
                  <span typeof="wsl:Message">
                         <strong>Parameters:</strong><br/>
                         <code>lat</code> (Required) <i>[integer]</i>
                         - the latitude of the location<br/>
                         <code>lon</code> (Required) <i>[integer]</i>
                         - the longitude of the location<br/>
                         <br/>
                         <strong>Parameters in Post:</strong><br/>
                         <code>hour</code> (Optional) <i>[integer]</i><br>
                         <code>min</code>  (Optional) <i>[integer]</i>
                         - the time of the forecast<br/>
                        </span>
                 </span>
                 <span rel="wsl:hasOutputMessage">
                  <span typeof="wsl:Message">
                         <strong>Output value:</strong>
                           weather and temperature of Location
                           Xml type
                           <a href="http://www.example.com/serviceID/responseSchema.xsd">
                             xmlSchema
                           </a>
                         </span>
                        </span>
                 </p>
```

```
                </span>
        </div>
</div>
```

Nell'esempio sopra ci riferiamo alle classi e alle proprietà RDF del modello 2.4 utilizzando i namespace *wsl* e *hr*. Nel primo blocco viene inserito il valore *wsl:service* che sta ad indicare che all'interno di quel blocco, sono contenute descrizioni di una o più operazioni ed eventuali label, esse vengono usate sul markup testuale per specificare delle etichette human-readable che verranno utilizzate per fornire informazioni all'utente.
Per indicare il blocco che contiene la descrizione di una singola operazione del Web Service, viene indicata dall'istanza *wsl:Operation* collegato dall'istanza padre identificato da *wsl:hasOperation*.
Per indicare l'indirizzo (URI) della risorsa su cui l'operazione opera, viene indicata dall'istanza *hr:hasAddress*. Se l'URI viene modellato modellato attraverso l'utilizzo dell'URITemplate, si inserisce l'attributo con valore *hr:URITemplate* che viene inserita spesso in un tag hyperlink, ma a volte, viene inserito in un tag *<code>*. All'indirizzo dobbiamo anche descrivere il metodo HTTP che si deve accompagnare la richiesta, e questa viene identificata dall'istanza *hr:hasMethod*, in caso di assenza di qualunque valore del metodo specificato, si prende come default il metodo GET.
Oltre all'URITemplate, bisognerà descrivere la sintassi ma anche la tipologia dei parametri che vengono utilizzati per la modellizzazione dell'URI. Queste informazioni vengono identificate mediante l'istanza di tipo wsl:Message all'interno di un blocco superiore *wsl:hasInputMessage* o *wsl:hasOutputMessage* a seconda se i parametri descritti dal blocco identificato da *wsl:Message* siano messaggi di input o di output.
In questo esempio, il messaggio di output è composto da un file XML e viene descritto mediante un link il quale si riferisce al Xml Schema del messaggio di output.

2.2.4 MicroWSMO

Il microformat hRESTS organizza la documentazione in HTML di un Web Service RESTful in maniera tale che essa sia facilmente predisposta per essere elaborata da un processo automatizzato. Il microformat identifica le chiavi o pezzi di informazioni che sono gia presenti nella descrizione, analogamente a quanto avviene per la creazione del WSDL il quale viene usato per Web services non RESTful.
L'hRESTS è un ottimo linguaggio di base che consente ulteriori estensioni per arricchire di informazioni la documentazione del servizio. Una estensione che si vuole presentare in questo sezione si chiama MicroWSMO [27], un linguaggio che consente di annotare concetti ontologici per indicare il significato semantico dei dati utilizzati nel servizio.

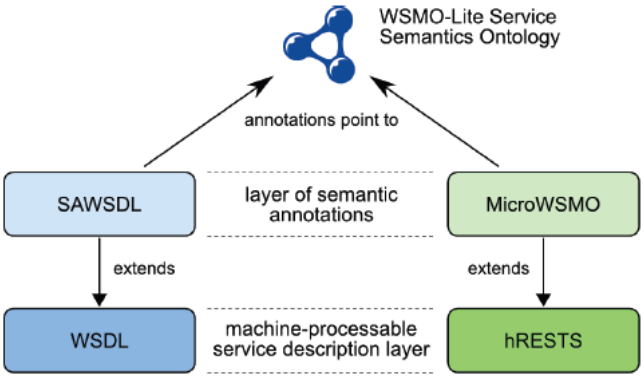


Figura 2.20: Modello funzionale di un servizio Web RESTful

Come l'hRESTS viene comparato all'utilizzo di WSDL, anche il microWSMO può essere visto come il SAWSDL, ovvero come ulteriore strato di descrizione che indica la semantica della sintassi. Non solo può essere paragonato al SAWSDL per la somiglianza dello scopo, ma anche perchè sfrutta gli stessi concetti e gli stessi meccanismi per annotare le ontologie.
Come il SAWSDL, anche il MicroWSMO definisce tre attributi XML mediante le proprietà dell'RDF:

- *modelReference* è usato su ogni componente nel modello del servizio per identificare il concetto mediante il suo appropriato URI.
- *liftingSchemaMapping* and *loweringSchemaMapping* sono utilizzati per associare le diverse trasformazioni che il messaggio deve subire, anch'esse vengono definite mediante URI.

La figura 2.20 illustra la relazione tra MicroWSMO e SAWSDL, insieme alla funzione di estensione alla descrizione delle specifiche del servizio. MicroWSMO è simile al layer SAWSDL solo che viene poggiato sopra all'hRESTS. Entrambi i linguaggi di semantica fanno riferimento alle stessa ontologia WSMO-Lite e, in questa maniera, permette di catturare la semantica dei servizi indipendentemente dallo strato tecnologico (WSDL/SOAP o REST/HTTP).
Riprendendo l'esempio sopra descritto, vediamo come annotare semanticamente le informazioni.

```
<div typeof="wsl:Service" about="#svc1"
xmlns:hr="http://www.wsmo.org/ns/hrests#"
xmlns:wsl="http://www.wsmo.org/ns/wsmo-lite#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:sawsdl="http://www.w3.org/ns/sawsdl#">
<h1>
        <span property="rdfs:label">Weather Forecast</span> service API
```

```
</h1>
        This is a service of
        <a rel="sawsdl:modelReference"
         href="http://www.soa4all.eu/models/weatherForecasts">Weather Forecast</a>
        <div rel="wsl:hasOperation">
         <span typeof="wsl:Operation" about="#op1">
        <h2>Operation <code property="rdfs:label">
        getForecastByLocation</code></h2>
        <p>Invoked using the method
        <span property="hs:hasMethod">POST</span> at
        <code property="hr:hasAddress"
        datatype="hr:URITemplate">
        http://example.com/rest/service/getForecastByLocation?{-join|&|lat,lon}
        </code><br/>
        <span rel="wsl:hasInputMessage">
        <span typeof="wsl:Message">
        <strong>Parameters:</strong><br/>
        <code>
        <a rel="sawsdl:modelReference"
         href="http://www.soa4all.eu/models/parameters/GPS#latitude">
        lat</a></code>
  (<a rel="sawsdl:modelReference"
          href="http://www.soa4all.eu/models/general/parameters#required">
         Required</a>)
        <i>[<a rel="sawsdl:modelReference"
          href="http://www.soa4all.eu/models/general/types#integer">
                integer</a>]</i> - the latitude of the location<br/>
        <code>
         <a rel="sawsdl:modelReference"
          href="http://www.soa4all.eu/models/parameters/GPS#longitude">
         lon</a>
        </code>
        (<a rel="sawsdl:modelReference"
          href="http://www.soa4all.eu/models/general/parameters#required">
        Required</a>)
        <i>[<a rel="sawsdl:modelReference"
         href="http://www.soa4all.eu/models/general/types#integer">
         integer</a>]</i> - the longitude of the location<br/>
        <strong>Parameters in Post:</strong><br/><code>
  <a rel="sawsdl:modelReference"
          href="http://www.soa4all.eu/models/parameters/time#timeDistance_hours">
        hour</a></code>
        (<a rel="sawsdl:modelReference"
          href="http://www.soa4all.eu/models/general/parameters#optional">
         Optional</a>)
   <i>[<a rel="sawsdl:modelReference"
          href="http://www.soa4all.eu/models/general/types#integer">
                integer</a>]</i><br>
               <code>
               <a rel="sawsdl:modelReference"
               href="http://www.soa4all.eu/models/parameters/time#timeDistance_minutes">
               min</a></code>
               (<a rel="sawsdl:modelReference"
               href="http://www.soa4all.eu/models/general/parameters#optional">
               Optional</a>)
               <i>[<a rel="sawsdl:modelReference"
```

```
                href="http://www.soa4all.eu/models/general/types#integer">
    integer</a>]</i>– the time of the forecast<br/>
                </span>
        </span>
        <span rel="wsl:hasOutputMessage">
         <span typeof="wsl:Message">
                <strong>Output value:</strong>  weather and temperature of Location
                 <a rel="sawsdl:modelReference"
                        href="http://www.soa4all.eu/models/general/mediatypes#text/xml">
                        Xml</a> type
                        <a rel="sawsdl:modelReference"
                        href="http://www.example.com/serviceID/responseSchema.xsd">xmlSchema</a>
                </span>
               </span>
        </p>
       </span>
 </div>
</div>
```

Come possiamo notare, vengono inseriti dei tag hyperlink con l'istanza modelReference del namespace SAWSDL impostato mediante il prefisso *sawsdl* dichiarato nel primo blocco. Ogni parametro nel messaggio di input viene annotato con il suo significato, viene fornita l'informazione sulla tipizzazione del dato, anche se è opzionale o richiesto. Tutte queste informazioni non era possibile fornirle mediante hRESTS.
Per annotare la semantica del messaggio di output, nel nostro esempio, viene annotato direttamente l'Xml-Schema.
Come sappiamo dalle sezioni precedenti, la maggior parte dei servizi RESTful viene descritta solo mediante pagine HTML che possono essere capite solo da utenti umani. Per poter far si che queste descrizioni vengano processate da calcolatori, dobbiamo convertire o estendere queste descrizioni con i linguaggi esposti nel capitolo 3. Nel progetto SOA4All si è scelto di optare per i linguaggi hRESTS, per la sintassi, applicando al di sopra un livello che descriva la semantica mediante il linguaggio MicroWSMO. Per facilitare questa annotazione, l'università KMI (Knowledge Media Institute) ha sviluppato per il progetto SOA4All un tool grafico chiamato SWEET [23] supportando l'utente ad annotare a livello semantico le caratteristiche del servizio.

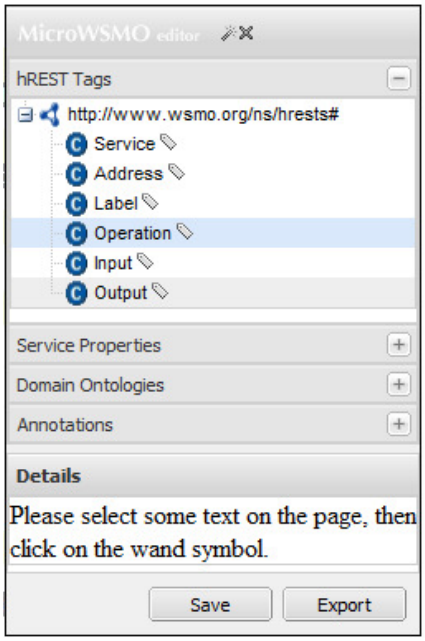


Figura 2.21: Widget SWEET

Le funzionalità che mette a disposizione questo tool sono:

- **Inserimento dei tag microformat hREST** in una pagina HTML che descrive un servizio RESTful marcando le sue proprietà principali (operazioni, URI, metodi HTTP, input, output e label)
- **Ricerca integrata di ontologie** per annotazione semantiche delle informazioni relative alle proprietà del servizio
- **Inserimento dei model reference MicroWSMO**, associati al significato semantico delle proprietà del servizio.
- **Salvataggio dell'HTML semanticamente annotato** della descrizione del servizio RESTful, il quale può essere ripubblicato sul Web
- **Estrazione del formato MicroWSMO in RDF** della descrizione del servizio, basato sull'HTML annotato

Il tool venne presentato nel 2009 al "Beyond SAWSDL Workshop" [26], e consiste in un widget creato in javascript che permette mediante la selezione del contenuto HTML di una pagina di applicare le varie descrizioni.
Un esempio dimostrativo lo si nota nella figura 2.22 che mostra come annotare un parametro che viene mostrato nella descrizione in una pagina HTML, mediante selezione, il widget riconosce la parola e consente di associare sia i tag hREST che descrivo la sintassi del parametro, sia l'ontologia per la parte semantica.

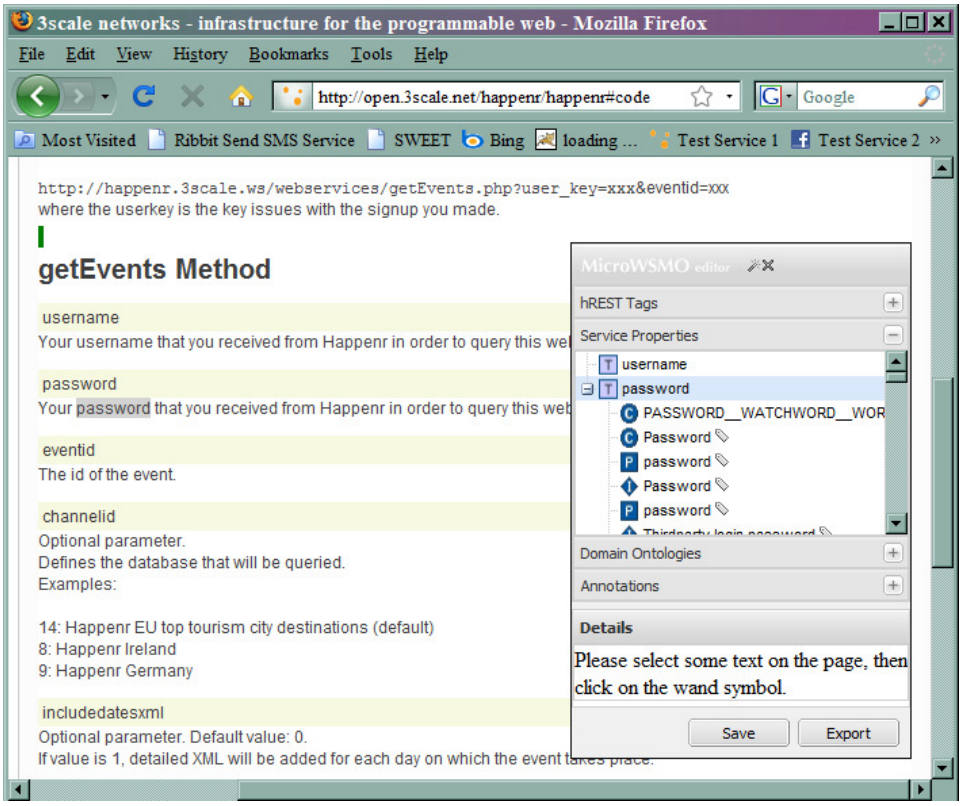


Figura 2.22: Esempio: Descrizione e annotazione della proprietà "Password"

2.2.5 SA-REST

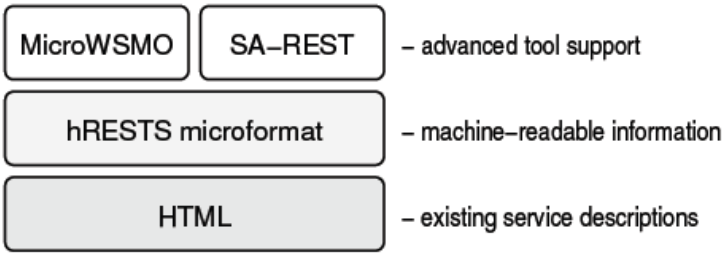


Figura 2.23: Modello funzionale di un servizio Web RESTful

MicroWSMO si appoggia su hRESTS ma utilizza concetti facendo riferimento sempre al *sawsdl* mediante la parola chiave *modelreference* che, alle volte, può essere troppo generico e non funzionale.
Esiste un altro linguaggio che consente di annotare la semantica e si appoggia ad hRESTS come si nota nella figura 2.23, questo linguaggio si chiama SA-REST [33].
Il linguaggio SA-REST è nato soprattutto perchè a differenza dei servizi SOAP, un servizio RESTful può maneggiare non solo documenti in formato Xml, ma anche altri formati di scambio dati, tra cui il più utilizzato in questa tipologia è il JSON, altri linguaggi come MicroWSMO, non si adattano a descrivere queste diverse proprietà. La necessità principale per cui è nata questa tipologia di linguaggio è da attribuire alla creazione e all'utilizzo di componenti mashup, ovvero applicazioni Web all'interno di altre applicazioni Web.
Gli sviluppatori preferiscono utilizzare servizi che sono omogenei sia nei formati di scambi sia per quanto riguarda le librerie applicative da utilizzare, per evitare inconvenienti sulla compatibilità. L'estensione SA-REST nasce proprio per questo, e tra le sue proprietà, ne contiene due che permettono di identificare il tipo di formato con cui scambiare i dati e indicare quali linguaggio di programmazione sono compatibili, come nell'esempio riportato qui sotto.

```
<div class=''service'' id=''svc''>
<p>The output format of the operations of the
<code class=''label''>ACME Hotels</code> service is
<span class=''data-format''>JSON</span>.
Client libraries are available in
<span class=''p-lang-binding''>Java</span> and
<span class=''p-lang-binding''>PHP</span>.
</p></div>
```

Questo esempio mostra l'utilizzo di due classi che identificano il formato (*data-format*) e il linguaggio di programmazione il cui formato è compatibile (*p-lang-biding*).

2.2.6 WADL

Il Web Application Description Language è un vocabolario XML che consente di descrivere le caratteristiche di una risorsa che un servizio RESTful espone. Questo linguaggio ha molte analogie con il WSDL utilizzato nei servizi SOAP, partendo da una somiglianza con il nome fino ad arrivare al risultato che si ottiene applicandolo.
Il WADL non è ancora utilizzato in maniera massiva per poterlo indicare come uno standard, a differenza del WSDL, poichè la maggior parte dei servizi RESTful vengono descritti in HTML perchè non sono pensati per una elaborazione automatizzata, e l'utilizzo del WADL consisterebbe nel convertire tutte le descrizioni in questo linguaggio, cosa che va contro i principi di semplicità d'utilizzo del servizio REST.
Per ogni risorsa che il web service tratta, può essere specificato un WADL e oltre alle caratteristiche sintattiche di esso, ne descrive la modalita con cui si può manipolarla. Supporta la descrizione dell'URI-Template e tutti i metodi HTTP. Una caratteristica molto importante che si mette in luce del WADL, è quella di favorire la descrizione del formato di rappresentazione della risorsa andando più in là della solita coppia chiave/valore. Si può descrivere diversi formati di rappresentazione, ad esempio si può descrivere una rappresentazione in forma XML facendo riferimento al suo Xml-Schema.
Il file ottenuto dal WADL per descrivere un servizio, si può suddividerlo in tre sezioni: la definizione della risorsa, la definizione del metodo utilizzato e la definizione della rappresentazione.
Per descriverlo nei dettagli, prendiamo come esempio un servizio esistente chiamato *www.icio.us* che consente di memorizzare i link internet come preferiti all'interno di una propria cartella personale.

Listing 2.5: Sezione con la definizione delle Risorse

```
<?xml version="1.0"?>
<application xmlns="http://research.sun.com/wadl/2006/07">
<!-- The resource -->
<resources base="http://api.del.icio.us/">

<doc xml:lang="en" title="The del.icio.us API v1">
Post or retrieve your bookmarks from the social networking website.
Limit requests to one per second.
</doc>

<resource path="posts">
  <resource path="recent">
   <method href="#getRecentPosts" />
  </resource>
</resource>

</resources>
...
</application>
```

Come primo elemento nella descrizione, viene indicato il percorso base dell'URI da cui partire per raggiungere le risorse, aggiungendo opzionalmente una breve descrizione del servizio. All'interno del blocco si identificano ogni singola risorsa indicando il percorso. Nel nostro esempio viene descritta la risorsa con l'URI *http://api.del.icio.us/posts/recent* e viene indicato che è possibile effettuare una richiesta HTTP su di essa (mediante il blocco *method*) e che le sue informazioni sono contenute nel documento in un blocco dal nome *getRecentPosts*.

Listing 2.6: Sezione con la definizione dei Metodi

```
<?xml version="1.0"?>
<application xmlns="http://research.sun.com/wadl/2006/07">
...
<!-- The method -->
<method id="getRecentPosts" name="GET">
  <doc xml:lang="en" title="Returns a list of the most recent posts." />
  <request>
   <param name="tag" style="form">
    <doc xml:lang="en" title="Filter by this tag." />
   </param>
   <param name="count" style="form" default="15">
    <doc xml:lang="en" title="Number of items to retrieve.">
     Maximum: 100
    </doc>
   </param>
  </request>
  <response>
   <representation href="#postList" />
    <fault id="AuthorizationRequired" status="401" />
  </response>
</method>
...
</application>
```

Nel frammento di XML riportato nel Listing 2.6 viene descritto il metodo (l'operazione) indicato nella descrizione della risorsa precedente (*getRecentPosts*), viene evidenziata l'informazione che la richiesta dovrà fornire mediante il metodo GET aggiungendola alla URI, che equivalgono a due parametri in formato querystring con nome *tag* e *count*. Si identifica che il secondo parametro deve essere obbligatorio e se non viene specificato nessun valore, verrà impostato quello di default pari a 15.
La rappresentazione dei dati di input viene fornita direttamente alla definizione dei parametri, ma possono venire definiti a parte come mostrato per il messaggio di risposta. come risposta possiamo avere due diversi messaggi, il primo è quello descritto dalla rapresentazione identificato dal nome *postList*, ma se la richiesta non andasse a buon fine, viene restituito il messaggio identificato dal tag *fault* che descrive uno stato di errore.

Listing 2.7: Sezione con la definizione delle Rappresentazione

```
<?xml version="1.0"?>
```

```
<application xmlns="http://research.sun.com/wadl/2006/07">
...
<representation id="postList" mediaType="text/xml" element="posts">
<param name="post" path="/posts/post" repeating="true" />
</representation>
...
</application>
```

Il messaggio di output che si avrà in risposta alla richesta, sarà in formato *text/xml* e non viene definito direttamente l'Xml-Schema ma viene indicata la struttura mediante percorsi in stile XPath. Il linguaggio WADL permette anche di inserire un file xsd e di utilizzare quest'ultimo per definire il formato di rappresentazione delle risorse, come mostrato nel Listing 2.8.

Listing 2.8: Sezione con la definizione delle Rappresentazione mediante xsd

```
<?xml version="1.0"?>
<!-- This is a partial bootleg WADL file for the del.icio.us API. -->
<application xmlns="http://research.sun.com/wadl/2006/07"
xmlns:delicious="https://api.del.icio.us/v1/posts.xsd">
<grammars>
<include "https://api.del.icio.us/v1/posts.xsd" />
</grammars>
...
<representation id="postList" mediaType="text/xml" element="delicious:posts" />
...
</application>
```

2.2.7 OWL-S/WADL

Per i servizi SOAP abbiamo esteso il linguaggio WSDL inserendo notazioni sematiche derivate da ontologia OWL-S (linguaggio ontologico OWL per il Service) e abbiamo introdotto il nuovo termine di SAWSDL.
Per quanto riguarda il servizio REST abbiamo appena finito di descrivere un linguaggio che è paragonabile al WSDL, ma vogliamo anche annotare questo nuovo linguaggio chiamato WADL, con annotazioni semantiche, purtroppo però i concetti ontologici che descrivono il modo con cui si interagisce con il servizio, non sono gli stessi per un servizio SOAP e REST, quindi abbiamo qualche difficoltà nell'utilizzare solo l'ontologia classica OWL-S anche per annotare un WADL.
Per questo motivo si è studiato l' estensione di OWL-S soprattutto per le istanze del *supports* che definisce le modalità di iterazione di un servizio REST. Questa nuova estensione definita viene nominata *RESTfulGrounding* [30].

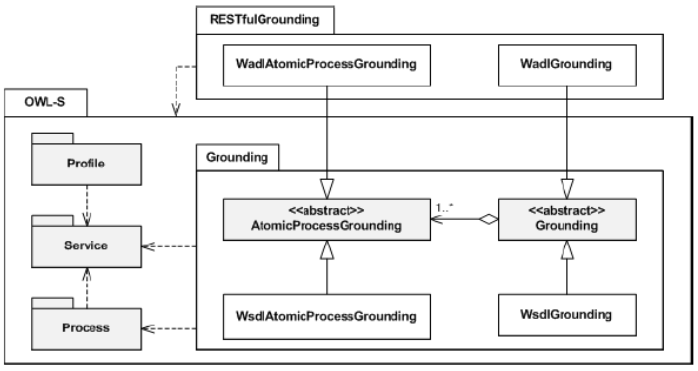


Figura 2.24: *RESTfulGrounding*: Estensione di OWL-S per OWL-S/WADL Grounding

OWL-S/WADL Grounding estende un layer astratto composto da due classi OWL-S, chiamate *Grounding* e *AtomicProcessGrounding*. La figura 2.24 è un UML class diagram che rappresenta questa estensione, e visualizza che le tre ontologie, *Service*,*Profile* e *Process*, che sono definite nella classica OWL, non sono state modificate ma rimangono sempre le stesse, e la figura vuole rafforzare l'idea che la nuova ontologia *RESTfulGrounding* non sostituisce il *Grounding* ma ne estende le potenzialità.
Un esempio di questa nuova ontologia è l'istanza *wadlResourceMethod* che consente di annotare un URI di una risorsa e il metodo HTTP che bisogna utilizzare sull'URI per effettuare una determinata operazione. Questo nella vecchia OWL-S non si poteva indicare per il fatto che il servizio SOAP non lo prevedeva. Oltre a questa proprietà, ce ne sono altre che non andremo a trattare ma che sono descritte nel documento riportato nella bibliografia.

```
<owl:ObjectProperty rdf:ID="wadlResourceMethod">
  <rdfs:domain rdf:resource="#WadlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="#WadlResourceMethodRef"/>
</owl:ObjectProperty>
```

Con l'utilizzo di questa nuova ontologia, si possono osservare due grosse limitazioni. La prima è che solo se si utilizza il WADL per descrivere a livello sintattico il servizio, è possibile annotare semanticamente con la nuova ontologia i concetti e nonostante che il WADL sia un linguaggio più che valido, non tutti i servizi REST vengono descritti in maniera formale.
La seconda limitazione è che il *RESTfulGrounding* supporta solamente un mapping 1:1 tra OWL-S e i parametri WADL.

2.3 Adattamento dei servizi

La composizione dei servizi rappresenta una combinazione di invocazioni dei servizi che consente all'utilizzatore del servizio di raggiungere un determinato obiettivo (goal). Per

poter combinare i servizi è necessaria l'interoperabilità di essi.
Adattare i servizi vuol dire far incontrare le richieste formulate dall'utilizzatore con i servizi forniti dal produttore, rendere cioè trasparenti al richiedente, le differenze delle interfacce che i servizi reali presentano nello scambio di messaggi, in quanto diversi agenti, possono cooperare scambiandosi messaggi per completare un task ma molto spesso l'eterogeneità a livello business e l'elevato numero di fornitori, ognuno dei quali può supportare differenti interfacce e protocolli, portano ad uno scostamento molto ampio tra risultato ottenuto e quello sperato.
Nel paper [7], viene proposto un approccio per poter adattare un servizio richiesto, con uno disponibile basandosi su tre elementi:

- *Mismatches* (disaccoppiamento): una serie di accoppiamenti non esatti tra il servizio astratto e concreto.
- *Mapping functions*: un numero componibile di funzioni di mapping che organizzate in script permettono di risolvere i mismatches identificati.
- *Runtime platform*: una piattaforma runtime che supporta il binding dinamico e l'adattamento interpretando lo script che è stato definito tra due servizi.

L'approccio prende in considerazione due servizi web e classifica due tipologie di disaccoppiamenti:

- Mismatches livello-interfaccia: riguarda le differenze tra i nomi delle operazioni tra servizio richiesto e disponibile e i parametri di queste operazioni. La classificazione include:
  - Differenze nei nomi delle operazioni: la richiesta formulata sul primo servizio prevede di invocare una certa operazione, ma il servizio invocato realmente ha un nome differente per questa operazione.
  - Differenze nei dati: i nomi dei parametri di input/output del servizio invocato dall'utilizzatore differiscono dal servizio reale, le differenze dei parametri (nome, tipo, numero, ordine).
- Mismatches livello-protocollo: riguarda le differenze nell'ordine in cui le operazioni del servizio astratto sono effettivamente invocate nel servizio concreto. Si distinguono in:
  - binding da uno a molti: caso in cui una operazione del servizio astratto può essere mappata in due o più operazioni del servizio concreto;
  - binding da molti a uno: due o più operazioni del servizio concreto possono essere mappate in una operazione del servizio astratto.

Sempre in [7] viene introdotto un linguaggio di mapping che consente di raggruppare tutte le informazioni permettendo di risolvere i *Mismatch* sopracitati. Questo *Mapping Script* viene creato sfruttando tutte le informazioni disponibili per attuare le seguenti funzioni:

ParameterMapping

: Risolve mismatches relativi ai dati. Questo operatore mappa un dato astratto ad uno concreto, in modo che il dato concreto possa essere usato invece del dato astratto quando vengono invocate operazioni equivalenti. Il mapping ovviamente è eseguito sia sul nome che sul tipo del dato.

ReturnParamenterMapping

: Stabilisce un collegamento tra i parametri di ritorno di una operazione del servizio astratto e concreto. Applicare questo operatore significa che nome, tipo e valore del parametro di ritorno dell'operazione concreta, sono mappati ai loro corrispettivi parametri di ritorno nella operazione reale.

OperationMapping

: Costruisce un collegamento tra una sequenza di operazioni astratte con una sequenza di operazioni concrete, considerate equivalenti.

StateMapping:

: Collega uno stato astratto con uno stato concreto. Effettuare questo collegamento significa che i due stati sono stati considerati equivalenti, laddove per equivalenza si può intendere, per esempio, che i due stati abbiano lo stesso nome.

TransitionMapping:

: Collega una transizione nel protocollo astratto ad una transizione nel protocollo concreto. Una transizione avviene se si ha a disposizione una coppia formata da uno stato e da una operazione da eseguire. Applicare questo operatore significa collegare la coppia stato-operazione del servizio astratto alla sua corrispondente nel servizio concreto.

Un progetto europeo denominato SeCSE (Service Centric System Engineering)[35], ha come obiettivo quello di realizzare una infrastruttura predisposta nel gestire l'ambiente dei servizi e per sostenere e gestire la composizione dinamica dei servizi che espongono interfacce sintatticamente non compatibili tra di loro, puntando sopratutto nell'effettuare l'adattamento in fase di runtime.
All'interno di questo progetto viene presentato e introdotto un framework denominato SCENE [10] che prevede la gestione dei servizi consentendo il discovery e la selezione di servizi compatibili con quello richiesto che, per esempio, potrebbe non essere disponibile

per ragioni, quali il livello di performance o altre definite negli SLA e consente il binding dinamico dal servizio richiesto al servizio compatibile disponibile.
Il progetto SeCSE ha realizzato una piattaforma in grado di gestire diverse fasi coinvolte con la gestione dei servizi Web. La fase dell'adattamento dei servizi, che è stata chiamata *adattamento sintattico*, si compone di una sequenza di operazioni:

1. Analisi dei mismatches esistenti tra le interfacce dei servizi;
2. Classificazione dei mismatches;
3. Realizzazione linguaggio di mapping;
4. Implementazione linguaggio di mapping;

Come punto di partenza si analizzano le interfacce dei servizi attraverso il WSDL degli stessi, si classificano i possibili mismatches e viene realizzato un linguaggio di mapping descritto in dettaglio in [7], che gestisce l'adattamento delle interfacce. Infine si passa all'implementazione del linguaggio di mapping, che viene realizzato nel componente chiamato *ServiceAdapter*, che è parte integrante della piattaforma SCENE realizzata nel progetto per la gestione dei servizi Web.
Un approccio di adattabilità a livello di interfaccia può avvenire anche a livello semantico, come mostrato in [9] e lo stesso approccio viene esteso con altre caratteristiche come descritto in [17]. Questo approccio introduce il SAWSDL, un estensione del linguaggio WSDL che include annotazioni semantiche per poter aumentare la precisione dell'adattabilità oltre al livello sintattico.
Lo strato semantico inserito, rispetto a quanto descritto in [9], subisce il rispetto dei vincoli per consentire la generazione dinamica di *Mapping Script*:

- Ogni operazione deve essere annotata con un attributo *modelReference*
- Lo schema WSDL deve essere annotato con un attributo *modelReference*
- Non deve essere usato l'attributo *loweringSchemaMapping*
- L'attributo *liftingSchemaMapping* può essere utilizzato in maniera opzionale
- Vengono utilizzati i tipi di dati definiti nell'XML Schema per definire tipi semplici

2.4 Adattamento SOAP-REST

Tutti gli approcci descritti nel paragrafo precedente, prendono in considerazione l'adattamento tra servizi in maniera generale e puntando soprattutto come riferimento alla tecnologia SOAP. Il Web 2.0 ha fornito un nuovo modo di vedere ed erogare servizi web, uno di queste tipologie viene chiamato REST.

Questa nuova tipologia ha avuto una grande diffusione grazie alla sua semplicità d'uso e al suo modo di interpretare le informazioni dei servizi come risorse. Il linguaggio standard utilizzato per descrivere la composizione dei servizi, è denominato BPEL e basato su servizi web descritti da WSDL che di solito sono implementati usando il SOAP ad oggi questo linguaggio non supporta i servizi REST dal momento che anche l'ultima versione del BPEL 2.0 si basa sul WSDL 1.1 che non definisce tutti i metodi che il protocollo HTTP mette a disposizione e che i servizi REST utilizzano.
In [46] viene presentato un framework che permette l'integrazione di servizi RESTful in un BPEL mediante una trasformazione automatizzata creando un WSDL del servizio REST il quale dovrà essere descritto mediante il formato WADL.
Il framework è composto da un componente denominato *Mashup Service Assembler* che si occuperà di trasformare il servizio RESTful in un servizio web SOAP creando un WSDL e permettendo al BPEL engine di poter comporre i servizi in maniera tradizionale e trasparente ed invocare il servizio di tipo RESTful sfruttando il componente.
Un altro approccio che si focalizza sull'integrazione dei servizi REST all'interno di una composizione descritta mediante il BPEL, la si trova nel paper [20] che presenta una best practices per ottenere un efficiente orchestrator di servizi eterogenei. L'approccio propone di suddividere la composizione in diversi BPEL e di creare due diversi motori, il primo *BPEL Engine* che si occupa di elaborare il processo BPEL e il secondo *Web orchestration Engine* che si focalizza sull'invocazione dei servizi interagendo mediante metodi nati per poter aumentare le performance.
Un'approccio che suggerisce una vera estensione del BPEL consentendo di avere un supporto nativo all'invocazione di un servizo RESTful, lo troviamo nell'articolo [32]. L'approccio descrive una possibile soluzione alternativa all'utilizzo del WSDL 2.0 per poter invocare i servizi RESTful, infatti, come detto precedentemente, la seconda versione del WSDL supporta la descrizione dei servizi REST. Il problema principale si trova nel BPEL che non ha pieno controllo dell'invocazione in quanto è il WSDL che contiene le descrizioni delle caratteristiche del servizio REST. La soluzione proposta analizza le varie caratteristiche e propone una estensione che consente al BPEL di poter invocare direttamente un servizio RESTful e di averne il pieno controllo a livello di composizione. Una proprietà fondamentale di questa estensione è quella di supportare l'intera evoluzione di stato della singola risorsa, rispettando in pieno i principi dell'architettura e supporta le diverse rappresentazione che la singola risorsa può assumere.
In [31] viene presentato un framework visuale, denominato "Jopera", che consente di creare una composizione di servizi RESTful permettendo di descrivere un intero processo basato su questo nuovo sistema di erogazione di servizi.
Una community molto attiva che ha come obbiettivo quello di fornire strumenti sempre più innovativi, è Apache ODE [29] che ha annunciato il rilascio di un WS-BPEL che tra le numerose caratteristiche, include una features che permette di invocare servizi web

REST-style.

CAPITOLO 3

APPROCCIO PROPOSTO

3.1 Analisi sulla compatibilità tra servizi

Per poter raggiungere il nostro obiettivo, dovremo andare ad affrontare un'analisi in cui verranno prese in considerazione le caratteristiche che le singole tecnologie mettono a disposizione per poter cercare il legame che le unisce.
Per poter adattare una richiesta o una risposta, qualunque sia la tecnologia di destinazione e di partenza, dobbiamo analizzare l'ammissibilità tra i singoli messaggi attraversando il flusso di comunicazione esistente tra il *Requestor* e il *Service Provider*.
Per definire la compatibilità con un'analisi generale, focalizziamoci sulla comparazione del comportamento nel quale due servizi vengono interrogati da un'utente, e la comunicazione consista in una sola richiesta formata da un singolo messaggio e un messsaggio rappresentante una singola risposta.
Presupponendo che entrambi i due servizi soddisfino lo stesso requisito, le richieste effettuate possono differire non solo dal tipo di struttura, ma anche dal numero di informazioni che il servizio ha bisogno per elaborare la richiesta. Riprendendo l'esempio precedente sul servizio metereologico, il primo servizio richiedeva solamente le coordinate geografiche per ottenere una risposta, ma se il secondo servizio che veniva invocato avesse avuto bisogno di ulteriori informazioni come l'orario, in quanto il servizio permette di ottenere l'informazione ad un livello più fine, l'adattamento che il *Service Adapter* eseguiva, non poteva più essere processato, dal momento che i parametri non erano sufficienti ad interrogare il *Service 2*.
Utilizziamo, per adesso, un approccio black-box sul componente *Service Adapter*, raffigurato in figura 3.1, sapendo solo che, in ingresso, richiede "a quale" servizio adattare e il messaggio su cui operare l'adattamento.

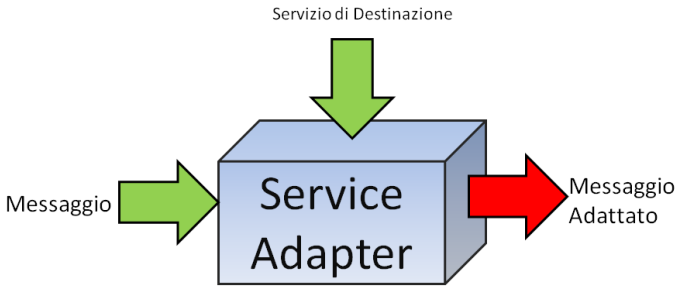


Figura 3.1: Componente Service Adapter

La prima analisi da effettuare, è quella di andare ad analizzare i casi della compatibilità dei parametri a livelo di obbligatorietà. La figura 3.2 rappresenta il risultato di questo primo studio. La tabella rappresentata, è stata compilata partendo dal presupposto che il messaggio di input al service adapter possa essere sempre adattato, infatti la terza riga che identifica il caso in cui il dato presente nel messaggio di input è opzionale rispetto a quello di output, l'adattamento non si può fare dato che non abbiamo sempre la certezza che il parametro del primo messaggio sia sempre presente.

Servizio 1	Servizio 2	Compatibile
Opzionale	Opzionale	OK
Obbligatorio	Opzionale	OK
Opzionale	Obbligatorio	NO
Obbligatorio	Obbligatorio	OK

Figura 3.2: Tabella sulla compatibilità dei parametri opzionali

Quello che si vuole raggiungere con il *Service Adapter*, è quello di garantire, all'utente o all'applicazione che lo sfrutterà, un'adattamento se i due servizi protagonisti vengono definiti compatibili tra di loro.
Sempre riferendoci alla figura 3.2, tralasciando i casi 1 e 4, la rimanente combinazione che vede nel messaggio di uscita un parametro opzionale e in quello d'ingresso lo stesso obbligatorio, la compatibilità è garantita, per il fatto che il servizio verrà sempre invocato con il parametro opzionale settato e l'invocazione, nel caso migliore, avrà successo.
Abbiamo sempre considerato i parametri contenuti nei messaggi in ingresso ai servizi, ma l'adattamento deve avvenire anche per i parametri contenuti nelle risposte dato che il nostro componente per nascondere l'adattabilità al *Requestor* deve essere in grado di rispondere con un messaggio avente la stessa struttura del messsaggio di risposta del *Service 1*, e le stesse considerazioni fatte precedentemente, devono valere anche per le risposte.
Da questa prima analisi possiamo incominciare a trarre le prime conclusioni;

> Due servizi sono compatibili (e quindi i loro messaggi sono adattabili) solo se le informazioni scambiate durante la comunicazione possono essere ritenute compatibili a livello di obbligatorietà.

I risultati mostrati, premettono che i parametri che vengono confrontati per verificarne l'adattabilità, esprimino lo stesso concetto semantico. In un ipotetico componente inserito in un'applicazione che implementi l'adattabilità, prima di applicare i risultati ottenuti, deve effettuare un confronto sulla semantica dei parametri in questione perchè non ci si può basare solo sul nome di esso, ma bisogna utilizzare strumenti più potenti per poterne identificare l'uguaglianza di significato, uno di questi strumenti sono i linguaggi di mark-up da applicare sui linguaggi descrittivi del servizio.

3.2 Adattabilità tra SOAP e RESTful

Le prossime analisi, verranno fatte considerando la tecnologia con cui sono creati i servizi, in quanto andremo a studiare quali sono i legami che esistono tra i diversi elementi che li compongono. Per prima cosa andiamo ad analizzare come mappare i diversi elementi che caratterizzano i servizi SOAP e quelli RESTful.

SOAP	REST
End Point	URITemplate
Nome Operazione	
NameSpace	Metodo (GET, POST,...)
XML	Parametri (URITemplate,body)
	XML in body
Risposta XML	Risposta XML

Figura 3.3: Tabella sul mapping tra SOAP e RESTful

La figura 3.3 riporta l'analisi di compatibilità tra un servizio SOAP e RESTful e indica l'associazione dei diversi elementi che compongono i singoli servizi. Come si nota dalla figura 2.16 un servizio che si basa sul protocollo SOAP, ha come elemento distintivo per invocare un'operazione rispetto ad un'altra, il nome dell'operazione inserito nel messaggio poichè l'URI, chiamato end-point, è sempre lo stesso. Partendo da questa considerazione, un'operazione SOAP viene identificata dalla coppia **End Point-Nome Operazione**, mentre nel servizio RESTful un'operazione corrisponde oltre che dall'URI che identifica la risorsa,anche dal metodo HTTP con cui si effettua la richiesta. Questa valutazione ci porta ad associare un'operazione SOAP in ingresso ad un'operazione RESTful in uscita dal nostro componente *Service Adapter*, mediante la coppia *End Point-Nome Operazione* e la coppia *URITemplate-Metodo*. Nel tabella riassuntiva nella colonna del servizio SOAP,

viene anche indicato il *NameSpace* come mapping insieme all'End Point e al Nome perchè, analizzando lo scenario riflesso a quello descritto, per ricostruire il messaggio SOAP da una richiesta REST, serve come informazione aggiuntiva anche il namespace.
Il messaggio di input di un servizio SOAP è sempre in un formato XML e le informazioni contenute in esso,vengono associate a parametri su un servizio RESTful in diversi modi. In questo servizio, i parametri di input possono essere forniti all'esecuzione dell'operazione sia attraverso l'elaborazione dell'URITemplate, sia attraverso il corpo della richiesta HTTP in formati diversi. Nel protocollo SOAP esistono quattro diverse tipologie di operazione, come illustrato nella figura 2.18, e vengono associati alla propria tipologia in base al loro comportamento.

	GET	POST	PUT	DELETE
OneWay Operation	NO	OK *	OK	OK
RequestResponse Operation	OK	NO*	NO	NO
Notification Operation	NON IMPLEMENTATE			
Solicit-Response Operation	NON IMPLEMENTATE			

Figura 3.4: Tabella sulla compatibilità della tipologia delle operazioni

La tipologia di operazioni che andiamo ad analizzare sono la *OneWay Operation* e la *RequestResponse Operation* dal momento che le due rimanenti, non sono principalmente utilizzate nei servizi SOAP, vengono citate perchè sono rimaste nella specifica per motivi storici, ma non si è riusciti a trovare ancora il meccanismo per implementarle su larga scala.
La *OneWay Operation* non può essere mappata solo sul metodo *GET*, come si nota in figura 3.4, in quanto il tipo di operazione SOAP consiste nell'inviare delle informazioni al servizio senza avere ulteriori informazioni al di fuori del messaggio di errore o di successo dell'esecuzione, invece il metodo GET è l'operazione opposta. Infatti quando si invoca su URI una risorsa il metodo *GET*, ci si aspetta sempre una risposta contenente la rappresentazione dello stato della risorsa. Per il metodo *POST*, è stato aggiunto un *, perchè non sempre l'invocazione del metodo *POST* su una risorsa, consiste nella creazione di una nuova istanza della risorsa, a volte viene utilizzato impropriamente al posto del *GET* a causa della somma dell'URI e dei parametri che vengono inviati mediante la querystring che non può superare il limite di 256 caratteri. Questa limitazione porta ad utilizzare il

POST per inviare delle informazioni all'interno del messaggio per evitare richieste incomplete. Nella progettazione del nostro lavoro, si è basati sui principi fondamentali della programmazione REST e quindi noi associamo i metodi HTTP ai comandi CRUD come abbiamo mostrato nella figura 2.9 e per noi l'operazione *OneWay* è adattabile sul *POST* e viceversa.
Il problema del metodo *POST*, si ripresenta anche per lo studio dell'adattabilità con l'operazione *RequestResponse*, ma come abbiamo premesso sopra, per noi il metodo *POST* non è compatibile con la tipologia di operazione perchè non deve restituire nulla come messaggi di risposta. Lo stesso discorso vale anche per gli altri metodi *PUT* e *DELETE*, cambia solo per il metodo *GET* in quanto è l'unica operazione che può essere adattata su questa tipologia di operazione.
Questi risultati però non dovrebbero pregiudicare l'adattabilità, in quanto tra i due servizi esiste l'adattatore e può anche modificare il comportamento di ogni singola operazione in quanto una operazione SOAP *RequestResponse* può essere adattato sul verbo *PUT*, solo se il messaggio di ritorno che si aspetta il servizio SOAP sia solo un messaggio di conferma o di errore, per la ragione che il protocollo HTTP è basato sul design pattern client-server e quindi un messaggio di ritorno lo si può sempre creare, in base alle informazioni che si possono trarre dalle descrizioni dei servizi. Per quanto riguarda l'adattabilità tra servizi REST oltre alla conversione e adattamento dei parametri contenuti nella richiesta e risposta, non si può giocare sulle combinazioni tra i vari metodi HTTP dato che, il nostro lavoro e le considerazioni precedenti si sono sempre fondate sul rispetto dei principi di design, che Roy Fielding ha espresso nella sua tesi introducendo il termine REST.

3.3 Considerazioni e semplificazioni

Bisogna aggiungere alcune considerazioni e semplificazioni, oltre a quelle già citate, sui risultati e sugli studi riportati sopra. Iniziamo con la compatibilità tra parametri contenuti nei messaggi. L'analisi è stata fatta sulla considerazione che i messaggi di request per entrambi i servizi si componessero dello stesso numero di parametri, ma questo non è sempre vero, infatti ci si potrebbe trovare in uno scenario in cui i parametri che permettono di comporre un messaggio adattato siano presenti su più messaggi che il *Requestor* deve inviare al servizio.
Si fa notare anche come le operazioni che abbiamo finora preso in considerazione per l'adattabilità, siano solo operazioni di lettura, ovvero, richieste di informazioni senza richiederne la modifica e la creazione, in quanto la problematica che si vuole affrontare in questo documento è l'analisi e la progettazione di un meccanismo che permetta l'adattabilità della singola richiesta, indipendentemente dallo scopo di essa, le problematiche dei parametri su più messaggi e quella della coerenza della tipologia di operazione, riguardano sviluppi futuri in quanto fanno parte di un discorso più generale e verrà descritto nell'ul-

timo capitolo.
Oltre al formato Xml che viene utilizzato nelle analisi, un servizio RESTful può utilizzare diversi formati come rappresentazione delle risorse. Viene preso in considerazione l'Xml visto che è il formato che viene utilizzato nel servizio SOAP ed è facilmente adattabile ai linguaggi di programmazione. Un altro formato diffuso che viene utilizzato nei servizi RESTful, è il formato JSON (JavaScript Object Notation) [1] ma esso può essere facilmente convertito ed elaborato come un Xml [2], quindi ci possiamo riferire alle stesse problematiche che abbiamo incontrato finora.
L'ultima considerazione e semplificazione che abbiamo adottato riguarda il protocollo di trasporto utilizzato per invocare le richieste. Abbiamo detto nel capitolo 2 che il protocollo SOAP poteva utilizzare qualsiasi protocollo per l'invocazione, l'importante è che chi invoca e il servizio risiedano sullo stesso canale. In queste analisi abbiamo dato per scontato che tutte le comunicazioni avvenissere sul protocollo HTTP per evitare il controllo della conversione dal canale di trasporto utilizzato dal SOAP all'HTTP dai servizi RESTful.

CAPITOLO 4

PROGETTAZIONE E IMPLEMENTAZIONE

Nel capitolo seguente verrà esposta la soluzione software sviluppata, sulla base degli studi e delle considerazioni riportate precedentemente. Verrà descritto un caso d'uso per fornire gli strumenti necessari alla comprensione della soluzione. Insieme verranno anche forniti esempi completi di servizi descritti mediante i linguaggi riportati nel documento e un'analisi dell'architettura software adottata e delle tecnologie introdotte.

4.1 Scenario di riferimento

Per poter spiegare e capire al meglio tutte le informazioni riportate in questo capitolo, vogliamo descrivere un particolare scenario e utilizzarlo per esprimere alcuni concetti fondamentali.
Riconsideriamo gli esempi che abbiamo citato nei capitoli precedenti, presupponiamo di avere due servizi implementati con le due tecnologie, SOAP e RESTful, e che permettono di raggiungere lo stesso scopo, quello di avere le previsioni meteo in una determinata località individuata dalle sue coordinate geografiche espresse in latitudine e longitudine.
Il servizio SOAP, richiede che nel messaggio di richiesta inviato al service, debbano essere presenti anche le informazioni orarie, in quanto il servizio restituisce solamente le previsioni ad una determinata ora.

Nome Parametro	Tipo	Annotazione Semantica
latitude	xs:double	GPS#latitude
longitude	xs:double	GPS#longitude
hour	xs:int	time#timeDistance_hours
min	xs:int	time#timeDistance_minutes

Tabella 4.1: Annotazione dei parametri del messaggio di richiesta SOAP

Nome Parametro	Tipo	Annotazione Semantica
utc	xs:string	time#UTCTime
wx	xs:string	weatherForecasts#WeatherConditionsIconURL
weather	xs:string	weatherForecasts#WeatherConditionsShortDescription
mslp	xs:string	weatherForecasts#Pressure_hPa
temp	xs:string	weatherForecasts#Temperature_Centigrades
relh	xs:string	weatherForecasts#Humidity_Percentage
prcp	xs:string	weatherForecasts#Precipitation_mm
wind	xs:string	weatherForecasts#Wind_Knots

Tabella 4.2: Annotazione dei parametri del messaggio di risposta SOAP

Nella tabella 4.1 riportiamo le annotazioni che sono riportate nel file SAWSDL (riportato interamente nell'appendice) che descrive gli elementi utilizzati nel servizio, basato su SOAP, e che esplicano la struttura dei messaggi utilizzati. Si può notare che le annotazioni semantiche che vengono applicate agli elementi più significativi, come si nota nel Listing 4.1, vengono utilizzati sia nei messaggi di input che in quelli di risposta.

Listing 4.1: Frammento di SAWSDL del servizio SOAP

```
<wsdl:portType name="TheWeatherCRFSoap">
 <wsdl:operation name="getForecast">
  <sawsdl:attrExtensions
   sawsdl:modelReference="http://www.soa4all.eu/models/
      weatherForecasts#WeatherForecastByTimeLocation" />
  <wsdl:input message="tns:getForecastSoapIn" />
  <wsdl:output message="tns:getForecastSoapOut" />
 </wsdl:operation>
</wsdl:portType>
```

Sempre dal Listing 4.1 si nota come il nome dell'operazione venga annotata semanticamente mediante l'URI

http://www.soa4all.eu/models/weatherForecasts#WeatherForecastByTimeLocation

favorendo una maggior lettura da parte di agenti autonomi e di poter associare la tipologia di operazioni che possono mettere a disposizione più servizi.

Anche il secondo servizio basato sui principi di progettazione REST, ha lo stesso scopo di fornire le previsioni meteo di una località identificata mediante le sue coordinate geografiche, ma al contrario del servizio SOAP, le informazioni orarie sono facoltative come mostrato nella tabella 4.3 in cui si riportano i valori semantici e sintattici che si incontrano nella descrizione, e i parametri orari sono contrassegnati come opzionali.
Nel Listing 4.2 viene mostrata un frammento della descrizione in hRESTS con annotazione MicroWSMO ottenuta utilizzando il tool SWEET su una pagina HTML in cui è esposto il servizio. Si evidenzia che sull'URITemplate del servizio, è annotata la stessa URI semantica che è stata utilizzata nel servizio SOAP per annotare il nome dell'operazione.

Listing 4.2: Descrizione hREST con MicroWSMO dell'operazione e dei parametri obbligatori

```
<div typeof="wsl:Service" about="#svc1"
 xmlns:hr="http://www.wsmo.org/ns/hrests#"
 xmlns:wsl="http://www.wsmo.org/ns/wsmo-lite#"
 xmlns:sawsdl="http://www.w3.org/ns/sawsdl#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
 <h1>
  <a href="servizio.html">
   <span property="rdfs:label">Weather Forecast</span>
  </a> service API
 </h1>
 This is a service of <a rel="sawsdl:modelReference"
     href="http://www.soa4all.eu/models/weatherForecasts">
      Weather Forecast</a>
 <div rel="wsl:hasOperation">
  <span typeof="wsl:Operation" about="#op1">
   <h2>Operation
        <code property="rdfs:label">
          getForecastByLocation
        </code></h2>
   <span property="hr:hasMethod">GET</span>
   <code property="hr:hasAddress" datatype="hr:URITemplate">
    <a rel="sawsdl:modelReference"
        href="http://www.soa4all.eu/models/operations/
              weather#WeatherForecastByTimeLocation">
        http://demo.cefriel.it/rest/previsioni/
             getForecastByLocation?{-join|&amp;|lat,lon,hour,min}
        </a>
   </code>
  [...]
</div>
```

Per indicare l'obbligatorietà dei parametri utilizzati nell'URITemplate, il linguaggio hRESTS non permette di indicare quest'informazione e per sopperire a questa mancanza viene utilizzata l'annotazione semantica impostando l'URI

http://www.soa4all.eu/models/general/parameters#required

come valore del *modelReference*, come viene mostrato nella tabella 4.3.

Parametro	Tipologia	Semantica	Obbligatorietà
lat	types#double	GPS#latitude	parameters#required
lon	types#double	GPS#longitude	parameters#required
hour	types#integer	time#timeDistance_hours	parameters#optional
min	types#integer	time#timeDistance_minutes	parameters#optional

Tabella 4.3: Annotazione dei parametri del servizio RESTful

Listing 4.3: Descrizione hREST con MicroWSMO del servizio

```
[...]
<span rel="wsl:hasOutputMessage">
 <span typeof="wsl:Message">
  <strong>Output value:</strong>  weather and temperature of Location
   in  <a rel="sawsdl:modelReference"
        href="http://www.soa4all.eu/models/general/mediatype#text/xml">xml</a>
   format type.
  <a rel="sawsdl:modelReference"
     href="http://demo.cefriel.it/previsioni/OperationIDOutputSchema.xsd">
     Xml Schema
  </a>
 </span>
</span>
[...]
```

Nel Listing 4.3 si mostra come nel file html non viene descritta la struttura del messaggio di risposta in maniera diretta come avviene nel WSDL, ma viene inserito un link ad un file che descrive l'Xml Schema del messaggio. Per descrivere l'informazione, viene inserita un'annotazione semantica che identifica il media type della risposta, in questo caso *http://www.soa4all.eu/models/general/mediatype#text/xml*, e il link del file xsd viene trasformato in un'annotazione semantica.
Per effettuare l'adattamento, anche il file Xml Schema dovrà essere semanticamente annotato per poter associare i concetti. Nella tabella 4.4 vengono mostrati i parametri con le loro annotazioni che si incontrano nel file di output nel quale viene descritta la struttura dei messaggi di risposta. Nella soluzione che andremo a descrivere, si impone che ogni operazione deve descrivere il messaggio di output in un file diverso.

Parametro	Tipologia	Semantica
temperatura	xs:string	weatherForecasts#Temperature_Centigrades
previsione	xs:string	weatherForecasts#WeatherConditionsShortDescription

Tabella 4.4: Annotazione dei parametri del servizio RESTful

Questa descrizione può essere processata da un compilatore in quanto il codice HTML ottenuto utilizza attributi e formati XHTML. Ma nel nostro caso, utilizziamo un formato più compatto; il codice XHTML viene processato mediante dei tool [45], come quello

fornito dalla W3c, per ottenere una descrizione in RDF con solo le informazioni necessarie al compilatore e quindi ripulire tutte quelle parti in HTML superflue che possono rendere difficoltosa la lettura da parte di un processo autonomo.
Infine, riportiamo anche un esempio di adattabilità sui messaggi validati in base alle strutture descritte in precedenza. Prendiamo come riferimento la richiesta per conoscere le previsioni meteo di una città localizzata con coordinate latitudine 45,4612 e longitudine 9,1878 alle ore 16:15. Il messaggio SOAP risultante che verrà inviato verso il nostro componente Service Adapter è riportato nel Listing 4.4.

Listing 4.4: Messaggio SOAP

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:it="http://it.crf.TheWeather/">
 <soapenv:Header/>
 <soapenv:Body>
  <it:getForecast>
   <it:latitude>45.4612</it:latitude>
    <it:longitude>9.1878</it:longitude>
    <it:hour>16</it:hour>
    <it:min>15</it:min>
   </it:getForecast>
 </soapenv:Body>
</soapenv:Envelope>
```

Il messaggio SOAP, a fronte di un errore di comunicazione del servizio interrogato o di un errore nell'elaborazione, verrà riadattato su una richiesta al servizio REST e il Service Adapter costruirà l'URI

http://demo.cefriel.it/rest/previsioni/
getForecastByLocation?lat=45.4612&lon=9.1878&hour=16&min=15

utilizzando i dati contenuti nel messaggio SOAP.

Listing 4.5: Risposta Rest

```
<?xml version="1.0" encoding="UTF-8"?>
 <previsioniMeteo>
 <temperatura>-6</temperatura>
 <previsione> clear, rain</previsione>
</previsioniMeteo>
```

Il servizio REST risponderà con un messaggio in formato Xml (Listing 4.5) con le informazioni metereologiche richieste, ma il nostro componente deve adattare anche la risposta al servizio SOAP (Listing 4.6), in quanto l'utente che ha inviato il messaggio, si aspetta una risposta strutturata come descritto dal WSDL del servizio.

Listing 4.6: Messaggio SOAP (2)

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <soap:Body>
  <getForecastResponse xmlns="http://it.crf.TheWeather/">
   <getForecastResult>
    <weather>clear, rain</weather>
    <temp>-6</temp>
   </getForecastResult>
  </getForecastResponse>
 </soap:Body>
</soap:Envelope>
```

4.2 Mapping Script

Il *Service Adapter* per effettuare l'adattabilità ha bisogno di sapere come creare il nuovo messaggio utilizzando i dati che sono presenti nella richiesta.
Un meccanismo atto a fornire queste informazioni è il *Mapping Script* il quale consiste in un file Xml in cui vi sono tutte le informazioni che permettono al nostro componente *Service Adapter* di poter convertire un messaggio di un servizio in una richiesta o risposta accettata dal secondo service.
Il *Mapping Script* viene citato nella tesi di De Giorgio [16] come strumento per mappare due servizi SOAP, in questo documento, verrà presentata una nuova versione con modifiche ed aggiunte che sono state apportate per consentire l'adattamento sulle diverse tipologie di web service.
Le informazioni che deve contenere questo file sono i risultati dell'analisi che abbiamo effettuato ed esposto nel capitolo precedente, infatti dovrà contenere indicazioni su:

- *operazione*, la traduzione tra un'operazione del servizio richiesto e l'operazione del servizio adattato.
- *input*, la struttura della richiesta che deve effettuare il *Service Adapter* con i relativi parametri del messaggio ricevuto.
- *output*, la struttura della risposta che deve restituire il *Service Adapter* con i relativi parametri contenuti nel messagio di risposta ottenuto.

Per facilitarne la costruzione, il file viene suddiviso in sezioni logiche e ci consente anche di poter elaborare i dati più facilmente. Utilizziamo gli stessi servizi dello scenario descritto e discutiamo il *mapping script* risultante che descrive l'adattamento da SOAP a REST.

Listing 4.7: Mapping Script: Header

```
<tns:Mapping xmlns:tns="http://www.soa4all.eu/MappingLanguageSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <tns:ExpectedNameSpace>
```

```
    http://it.crf.TheWeather/
  </tns:ExpectedNameSpace>
  <tns:ExpectedEndpoint>
    http://inrete.dyndns.info/TheWeather/TheWeatherCRF.asmx
  </tns:ExpectedEndpoint>
  <tns:AvailableDescription>
    http://demo.cefriel.it/rest/services/WeatherForecast.htm
  </tns:AvailableDescription>
  [...]
</tns:Mapping>
```

Nel *Mapping Script* si fa riferimento ai servizi da adattare come Servizio Atteso (*Expected*) e Servizio Disponibile (*Available*). Per servizio *Expected* si intende il servizio dal quale l'utente si aspetta che risponda quando invia il messaggio verso il service adapter, mentre il servizio *Available* è il servizio su cui bisogna fare l'adattamento e che fornirà realmente i messaggi.
Nel Listing 4.7 viene descritto l'header del file e fornisce le prime informazioni necessarie ai servizi. In base a questi tag, il componente che effettua il parser, può già capire quale sia il tipo di adattamento da effettuare. Per un servizio REST i tag che sono presenti nell'header, descrivono solo il link alla descrizione human-readable del servizio e il nome che può assumere è *ExpectedDescription* o *AvailableDescription* in base se il servizio REST è quello "Atteso" o quello "Disponibile".
Per il servizio di tipo SOAP, i tag presenti indicano l'URI dell'end-point e l'eventuale Namespace da utilizzare all'interno dell'Xml dei messaggi. Queste informazioni sono definite mediante la coppia di tag *ExpectedEndPoint - ExpectedNameSpace* o *AvailableEndPoint - AvailableNameSpace* sempre in base a quale sia il servizio. Ovviamente la presenza di questi tag permette di riconoscere quale tipo di adattabilità riporta il *Mapping Script*.

Listing 4.8: Mapping Script: OperationRename

```
[...]
 <tns:InterfaceRule ID="weather2previsioni">
  <tns:OperationRename ID="weather2previsioni">
   <tns:ExpectedService>
    <tns:OperationName>
          getForecast
        </tns:OperationName>
   </tns:ExpectedService>
   <tns:AvailableService>
    <tns:URITemplate>
      http://demo.cefriel.it:3000/rest/previsioni/
      getForecastByLocation?{-join|&amp;|lat,lon}
    </tns:URITemplate>
    <tns:Method>POST</tns:Method>
   </tns:AvailableService>
  </tns:OperationRename>
 </tns:InterfaceRule>
[...]
```

L'adattamento di un messaggio da un servizio ad un altro, comporta molto spesso alla modifica del nome dell'operazione, tra SOAP e SOAP, e tra REST e SOAP la conversione è molto più drastica in quanto, come abbiamo già detto, l'operazione SOAP si mappa su una URI del servizio REST. Nel Mapping Script è presente una sequenza di tag chiamati *InterfaceRule* nel quale viene riportata l'operazione da effettuare. Questi tag vengono identificati dall'attributo *ID* e la sua funzione verrà descritta più avanti.
All'interno del tag mostrato nel Listing 4.8, viene inserita l'operazione di adattabilità a livello di operazione mediante il tag *OperationRename*. Sempre con la stessa logica descritta per l'header, abbiamo due tag contenitori denominati *ExpectedService* e *AvailableService* i quali contengono, coerentemente all'header, le informazioni sull'operazione diversificando la tecnologia su cui si basano. Per i servizi SOAP viene indicato il nome dell'operazione dal tag *OperationName*, mentre per i servizi REST viene indicato l'URI (anche in formato URITemplate) della risorsa mediante il tag *URITemplate* e il metodo HTTP da invocare sulla risorsa, dal tag *Method*.

Listing 4.9: Mapping Script: DataTypeRename

```
[...]
 <tns:InterfaceRule ID="weather2previsioni_0">
  <tns:ReferenceRuleID>
     weather2previsioni
  </tns:ReferenceRuleID>
  <tns:DataTypeRename total="false">
   <tns:Wrapper>
    <tns:ExpectedService>
     <tns:Name>getForecast.latitude</tns:Name>
     <tns:Type>double</tns:Type>
    </tns:ExpectedService>
    <tns:AvailableService>
     <tns:Name>lat</tns:Name>
     <tns:Type>double</tns:Type>
    </tns:AvailableService>
   </tns:Wrapper>
   <tns:Wrapper>
    [...]
   </tns:Wrapper>
   [...]
  </tns:DataTypeRename>
  <tns:ReturnMappingRuleID>
     weather2previsioniRet
  </tns:ReturnMappingRuleID>
 </tns:InterfaceRule>
[...]
```

Per descrivere l'adattamento dei messaggi con l'associazione dei parametri inseriti nel messaggio di input, vengono inseriti all'interno del tag *InterfaceRule* un tag chiamato *DataTypeRename*. In questo tag viene inserito una sequenza di informazioni identificate da *Wrapper*, in cui sono esposti i nomi dei parametri e il loro tipo.
Per reperire l'informazione all'interno del messaggio in ingresso al service adapter, nei tag

viene indicato il percorso in stile XPath ma separandoli mediante il segno "punto".
Il tag *ReferenceRuleID*, permette di indicare quale sia l'operazione da applicare prima di operare la trasformazione indicata all'interno del tag *InterfaceRule*, con la sua corrispondenza come attributo ID dello stesso tag. Questo identificativo viene utilizzato anche per contraddistinguere quale sia l'operazione da applicare sul messaggio di ritorno, infatti possiamo considerare l'adattabilità come una catena di operazioni, e in fondo a questa catena troviamo un tag denominato *ReturnMappingRuleID* nel quale è presente la chiave che segna l'inizio della sequenza di azioni da applicare al messaggio di risposta del servizio Disponibile e reinviarlo all'utente sottoforma di messaggio del servizio Atteso.
Questo formato di scripting è molto flessibile anche a livello di implementazione, infatti permette di estendere qualsiasi tipo di operazione che può essere eseguita sui dati per operare l'adattabilità. Un esempio facilmente riscontrabile sarebbe quello di avere un servizio che ritorna una sequenza di informazioni, come un negozio on line che fornisce l'elenco dei suoi prodotti. In questo caso, viene inserito all'interno del tag *DataTypeRename* al parametro che contiene le informazioni ripetute, un attributo il quale indica che quel tag sarà presente molte volte e dovrà essere adattato solo i parametri indicati nel mapping script. Nel Listing 4.10 viene rappresentato un esempio in cui un negozio on line restituisce un elenco di prodotti con tante informazioni, e il servizio Atteso ne restituisce solo alcuni di questi.

Listing 4.10: Mapping Script: Esempio di sequenza di informazioni

```
[...]
<tns:InterfaceRule ID="product2ProdottiRet">
 <tns:DataTypeRename total="false">
  <tns:Wrapper>
   <tns:ExpectedService>
    <tns:Name>getProductListResponse</tns:Name>
    <tns:Type>getProductListResponse</tns:Type>
   </tns:ExpectedService>
   <tns:AvailableService>
    <tns:Name>getProductListResponse</tns:Name>
    <tns:Type>getProductListResponse</tns:Type>
   </tns:AvailableService>
  </tns:Wrapper>
        <tns:Wrapper>
   <tns:ExpectedService>
    <tns:Name>getProductListResponse.return</tns:Name>
    <tns:Type maxOccurs="unbounded">return</tns:Type>
         </tns:ExpectedService>
   <tns:AvailableService>
    <tns:Name>getProductListResponse.return</tns:Name>
    <tns:Type>return</tns:Type>
   </tns:AvailableService>
        </tns:Wrapper>
        <tns:Wrapper>
   <tns:ExpectedService>
    <tns:Name>getProductListResponse.return.name</tns:Name>
    <tns:Type>string</tns:Type>
```

```
    </tns:ExpectedService>
    <tns:AvailableService>
     <tns:Name>getProductListResponse.return.name</tns:Name>
     <tns:Type>string</tns:Type>
    </tns:AvailableService>
   </tns:Wrapper>
          <tns:Wrapper>
    <tns:ExpectedService>
     <tns:Name>getProductListResponse.return.price</tns:Name>
     <tns:Type>float</tns:Type>
    </tns:ExpectedService>
    <tns:AvailableService>
     <tns:Name>getProductListResponse.return.price</tns:Name>
     <tns:Type>double</tns:Type>
    </tns:AvailableService>
          </tns:Wrapper>
  [...]
 </tns:DataTypeRename>
</tns:InterfaceRule>
[...]
```

Per indicare che il parametro del messaggio non è un valore primitivo ma è un tag contenitore in cui sono presenti dei valori primitivi, viene indicato come *Type*, lo stesso nome del tag in maniera tale che sia diverso dal tipo di dato primitivo che viene utilizzato negli Xml. Un'ultima osservazione per completare la descrizione del *Mapping Script*, nell'operazione di adattabilità del messaggio di ritorno, deve essere sempre indicato l'elemento root della struttura Xml, altrimenti si perderebbe l'informazione principale della struttura e l'eventuale namespace non concorderebbe con tutta la struttura.

4.3 Architettura

In questo lavoro, si è progettato un'architettura che permetta di implementare tutti gli studi e le proposte che sono state presentate nelle sezioni dei capitoli precedenti.
Quando si analizzano operazioni da effettuare sulla rete, bisogna sempre tenere conto delle latenze e delle performance che devono avere le web application.
Il nostro lavoro, si interpone tra una richiesta effettuata da un utente, e la sua risposta, quindi i tempi di elaborazione devono rientrare in certi limiti. Per questo motivo, l'architettura è stata creata per supportare due diversi processi: quello di elaborazione e creazione di informazioni contenute nel Mapping Script e il processo di adattamento dei singoli messaggi.
Questa suddivisione è stata fatta in quanto, dopo diverse prove effettuate, ci siamo resi conto che l'analisi delle caratteristiche dei servizi occupava molto più tempo dell'adattamento del singolo messaggio avendo le informazioni già disponibili. Per questo motivo, abbiamo inserito nella stessa architettura i due componenti separati. Abbiamo riscontrato che il processo che occupa più risorse e più tempo di elaborazione corrisponde al meccanismo di analisi e confronto dei diversi parametri, infatti, il tempo di elaborazione per la

creazione del mapping script aumenta esponenzialmente in proporzione al numero di informazioni che un servizio trasporta all'interno di un messaggio. Il motivo fondamentale è dato dal numero elevato di complessità a livello computazionale che possiamo ottenere con un numero di parametri molto alto.
Un altro processo che peggiora le performance della creazione del mapping script, è il motore di inferenza semantica. Questo problema di prestazioni è noto, in quanto l'inferenza viene fatta su un albero di diverse ontologie, per questo motivo abbiamo optato, nelle nostre prove e nelle nostre demo, un controllo solo a livello di singola ontologia di riferimento il quale è stata creata appositamente per lo scopo.

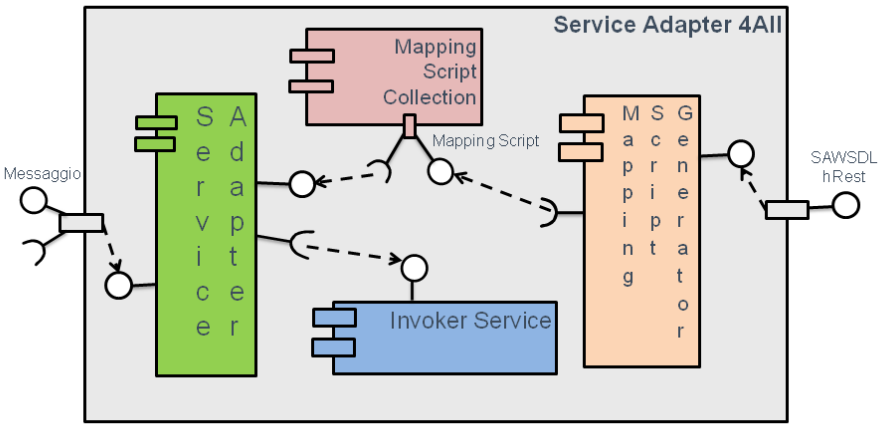


Figura 4.1: Architettura proposta

L'architettura proposta, viene raffigurata mediante la figura 4.1 la quale mostra l'organizzazione e i componenti che compongono la nostra soluzione. L'architettura è abbastanza semplice, si compone solo di tre parti software.
Il ruolo del componente *Service Adapter* è già stato descritto in precedenza e si può definire come l'access point della nostra architettura da parte dell'utente in quanto sarà questo componente ad esporre le API al nostro attore per poter richiedere l'adattamento e l'invocazione del servizio.

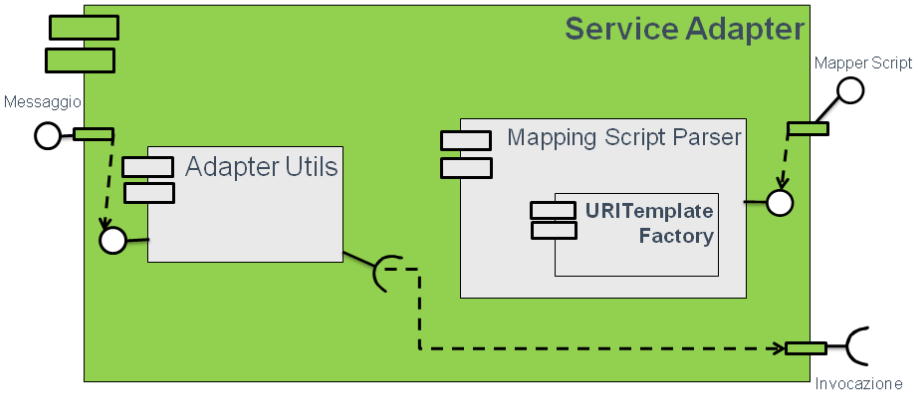


Figura 4.2: Service Adapter

Guardandolo a livello componentistico, il *Service Adapter* trova il suo oggetto principale nel *AdapterUtils* dove si trovano tutte le funzioni e gli strumenti atti ad adattare i messaggi che riceve. Il *Mapping Script Generator* legge il Mapping Script associato all'adattamento che si vuole effettuare ed insieme al precedente abilita l'adattamento. L'ultimo oggetto di cui si compone il nostro Adapter, viene utilizzato per generare e leggere gli URITemplate, non essendo ancora uno standard ma solo una proposta, non esiste nessun framework, ad oggi, che supporti questo formato.
Nell'architettura è stata inserita anche un *Invoker Service* che consente di invocare i servizi con le informazioni fornite dal *Service Adapter*. Questo componente è stato inserito per poter fornire ai componenti interni, una interfaccia comune per poter invocare i servizi, permettendo di avere un livello di astrazione nell'utilizzo dei servizi, in modo tale da consentire di alleggerire la gestione dei livelli protocollari che ogni tipologia di servizio mette a disposizione delegando ai diversi framework questo compito.

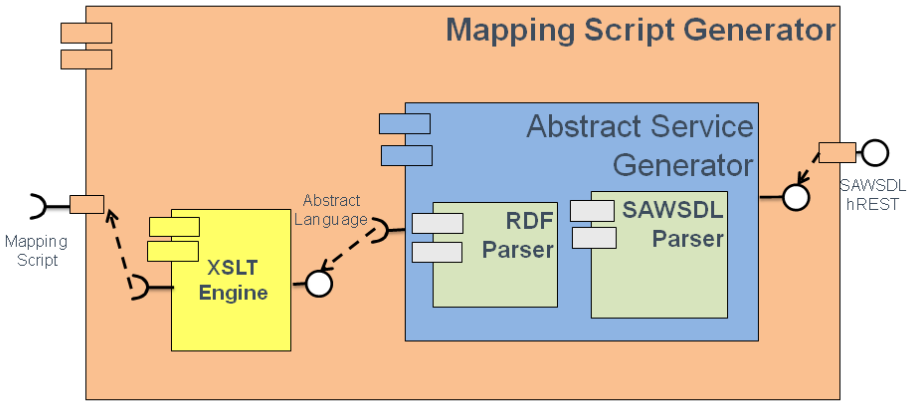


Figura 4.3: Mapping Script Generator

Nella figura 4.2, notiamo che il *Service Adapter* comunica con un datastore. Questo è un raccoglitore di Mapping Script che viene interrogato dall'Adapter per effettuare l'adattamento. Questo file, viene generato dal *Mapping Script Generator* , raffigurato nella figura 4.3. Questo componente è formato principalmente da un oggetto, chiamato *Abstract Service Generator* con il quale viene creata una rappresentazione intermedia delle descrizioni dei servizi. All'interno del *Abstract Generator* troviamo i componenti che ci permettono di leggere (effettuando il parsing) i file descrittori dei servizi, *SAWSDL Parser* per il SOAP e *RDF Parser* per i servizi REST.
Questo componente fornisce un'astrazione, per generare il mapping script e con questo meccanismo, noi riusciamo a superare le diversità che i file con il quale vengono descritti le funzionalità dei servizi, hanno tra di loro. Il processo di creazione permette di estrapolare le informazioni base che ci permetteranno di effettuare l'analisi di compatibilità e di renderli in un formato unico, sia per i servizi SOAP che per i RESTful in maniera tale da confrontare ed analizzare un solo formato e rendere semplice il confronto. Questo formato viene chiamato *Abstract Service Language* ed anch'esso è in formato Xml.
Come rappresentato nella figura 4.3, che illustra il componente, dopo la creazione del file da parte del'*Abstract Service Generator*, viene trasformato nel Mapping Script, mediante la potenza del XSLT, che consente, mediante espressioni regolari e diverse regole logiche, di confrontare due xml e di crearne uno nuovo. Nel nostro caso otterremo il nostro Mapping Script.

CAPITOLO 5

CONCLUSIONI

I servizi costruiti sul protocollo SOAP, possono affidarsi alla rigidità delle specifiche del protocollo e la sua applicazione storica permette di avere la certezza che le regole che costituiscono la sua costruzione non subiranno grosse modifiche. I servizi web che si basano sui principi di design REST non sono regolamentati da specifiche rigide e questo può portare al cambiamento continuo di utilizzo e ad una esplosione di modalità d'uso. Per questo motivo la maggior parte delle analisi esposte nel documento, potrebbero nel tempo non trovare più riscontro.
Il motivo della grande diffusione, in questi ultimi anni, di servizi RESTful è data dalla sua semplicità di utilizzo e il modello su cui poggia è di uso comune (HTTP). Chi vuole costruire un servizio REST, non ha bisogno di sapere come si costruisce un WSDL, i requisiti minimi sono quelli di saper costruire una pagina Web ben leggibile dall'utente umano. Per questo, quando si lavora e si prende in considerazione i servizi web RESTful, bisogna fare attenzione a non appesantirlo troppo, con molte regole complicate, altrimenti decade il principale vantaggio di utilizzo. Come accennato nel capitolo 3, il nostro progetto prende solo in considerazione messaggi confrontabili direttamente e singolarmente. Il mapping script progettato, consente di adattare un singolo messaggio di un servizio su un altro, senza specificarne il significato e senza analizzarne la coerenza.
Nell'articolo [7], vengono catalogati due tipologie di mismatch che si possono ottenere confrontando due servizi:

- *Mismatch a livello interfaccia*
- *Mismatch a livello di protocollo*

Il primo tipo di mismatch è quello analizzato nel nostro documento e riguardano le differenze tra i nomi delle operazioni e i parametri di esse. La classificazione include:

- Differenze nei nomi delle operazioni: la richiesta formulata sul servizio atteso prevede di invocare una certa operazione, ma il servizio disponibile ha un nome differente per questa operazione.
- Differenze nei dati: i nomi dei parametri di input/output del servizio atteso differiscono dal servizio disponibile, le differenze dei parametri (nome, tipo, numero, ordine).

Il secondo tipo di mismatch riguarda l'ordine con cui le operazioni vengono effettuate cercando di rispettare determinati criteri. Questo approccio viene citato in [8] e ha come idea centrale quella di considerare la sequenza di operazioni eseguite su un web service come una macchina a stati e considerare l'adattamento non delle singole operazioni, ma del singolo cammino che porta al raggiungimento dell'obiettivo prefissato. Un esempio di mismatch protocollare che si può riscontrare sopratutto in un servizio RESTful, è quello di eseguire operazioni di aggiornamento o di inserimento di una nuova risorsa su un determinato servizio ma esse non vengono eseguite su di esso. Prendiamo in considerazione un servizio web di storage. Mediante un servizio RESTful, possiamo gestire i singoli file (risorse) sul servizio eseguendo le usuali operazioni di cancellazione, modifica e creazione. Pensiamo di inviare una determinata richiesta al servizio per creare un file e questa interrogazione viene intercettata dal nostro Adapter. Se il servizio non fosse disponibile, la nostra attuale architettura dirotterebbe il messaggio verso un altro servizio, creando il file in uno storage diverso da quello atteso.
Se l'utente inviasse una serie di operazioni su questo file oltre alla creazione, e il servizio tornasse disponibile, il service adapter invierebbe il messaggio al servizio corretto ma l'utente si vede ritornare un messaggio di errore di tipo *File Not Found* che non si aspetta in quanto per lui la richiesta di creazione è andata a buon fine.
Con questo esempio, possiamo concludere con l'asserire che il prossimo passo per ottenere un completo adattamento di un servizio, è quello di considerare non solo il singolo messaggio, ma l'intera sequenza di operazioni che si possono effettuare e l'ordine con cui devono essere eseguite per elaborarle.

BIBLIOGRAFIA

[1] Introducing json. `http://www.json.org`.

[2] Json: The fat-free alternative to xml. `http://www.json.org/xml.html`.

[3] Danilo Ardagna, Marco Comuzzi, Enrico Mussi, Barbara Pernici, and Pierluigi Plebani. Paws: A framework for executing adaptive web-service processes, 2007.

[4] Tim Berners-Lee. Universal resource identifiers – axioms of web architecture. `http://www.w3.org/DesignIssues/Axioms`.

[5] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[6] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation, 2005.

[7] Luca Cavallaro and Elisabetta Di Nitto. An approach to adapt service requests to actual service interfaces. In *SEAMS 08 Workshops*, Leipzig, May 2008. ACM.

[8] Luca Cavallaro, Elisabetta Di Nitto, and Matteo Pradella. An automatic approach to enable replacement of conversational services. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, *ICSOC/ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 159–174, 2009.

[9] Luca Cavallaro, Gianluca Ripa, and Maurilio Zuccalà. Adapting service requests to actual service interfaces through semantic annotations. In *To appear in PESOS '09 Workshop*, 2009.

[10] Massimiliano Colombo, Elisabetta Di Nitto, and Marco Mauri. Scene: a service composition execution environment supporting dynamic changes disciplined through rules. In *LNCS*, pages 191–202. Springer Berlin / Heidelberg, 2006.

[11] Florian Daniel and Barbara Pernici. Web service orchestration and choreography: Enabling business processes on the web. *IGI Global*, 2008.

[12] Giovanni Denaro, Mauro Pezze, and Davide Tosi. Designing self-adaptive service-oriented applications, 2007.

[13] G. Meredith S. Weerawarana E. Christensen, F. Curbera. Web services description language (wsdl) 1.1. `http://www.w3.org/TR/wsdl`.

[14] Thomas Erl. *Service-Oriented Architecture Concepts, Technology, and Design*. Prentice Hall Professional Technical Reference, 2005.

[15] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. Master's thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[16] Teodoro De Giorgio. Il problema dell'adattamento nella composizione dei servizi: progettazione e sviluppo di una proposta di soluzione. Master's thesis, Politecnico di Milano, 2008.

[17] Teodoro De Giorgio, Gianluca Ripa, and Maurilio Zuccalà. Sawsdl for self-adaptive service composition. In Robert Meersman, Pilar Herrero, and Tharam S. Dillon, editors, *OTM Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 907–916. Springer, 2009.

[18] W3C Working Group. Simple object access protocol (soap) 1.1. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383487`.

[19] W3C Working Group. Web services architecture. `http://www.w3.org/TR/ws-arch/`.

[20] Kejing He. Integration and orchestration of heterogeneous services. In *Joint Conference on Pervasive Computing JCPC (2009)*.

[21] M. Nottingham J. Gregorio, M. Hadley. Uri template. `http://tools.ietf.org/html/draft-gregorio-uritemplate-03`.

[22] Karthik Gomadam Jacek Kopeck'y and Tomas Vitvar. hrests: an html microformat for describing restfulweb services. *IEEE Internet Computing*, 2008.

[23] SOA4All Knowledge Media Institute The Open University. Sweet - restful services annotation with sweet. `http://sweet.kmi.open.ac.uk/index.html`.

[24] Kopecký, Vitvar, Bournez, and Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE Internet Computing*, 11(6):60–67, 2007.

[25] Xitong Li, Yushun Fan, and Feng Jiang. A classification of service composition mismatches to support service mediation, 2007.

[26] Carlos Pedrinaci John Domingue Maria Maleshkova, Laurian Gridinoc. Supporting the semi-automatic acquisition of semantic restful service descriptions. `http://sweet.kmi.open.ac.uk/pub/SupportingSemi-AutomaticAcquisitionofSRS.pdf`.

[27] Jacek Kopeck'y Maria Maleshkova and Carlos Pedrinaci. Adapting sawsdl for semantic annotations of restful services. *IEEE Internet Computing*, 2009.

[28] IANA Internet Assigned numbers Authority. Uniform resource identifer (uri) schemes. `http://www.iana.org/assignments/uri-schemes.html`.

[29] Apache ODE. Project website. `http://ode.apache.org`.

[30] Maria Alice Grigas Varella Ferreira Otávio Freitas Ferreira Filho. Semantic web services: A restful approach. *IADIS International Conference WWW/Internet 2009*, 2009.

[31] Cesare Pautasso. Composing restful services with jopera. In *Software Composition*, pages 142–159, 2009.

[32] Cesare Pautasso. Restful web service composition with bpel for rest. *Data Knowl. Eng.*, 68(9):851–866, 2009.

[33] Amit P. Sheth, Karthik Gomadam, and Jon Lathem. Sa-rest: Semantically interoperable and easier-to-use services and mashups. *IEEE Internet Computing*, 11(6):91–94, 2007.

[34] Rudi Studer, Stephan Grimm, and Andreas Abecker. *Semantic Web Services*. Springer Xpert.press, Berlin, 2007.

[35] SeCSE Team. Secse project website. `www.secse-project.eu`.

[36] SOA4All Team. Soa4all project website. `www.soa4all.eu`.

[37] SOA4All Team. D11.1.1 oasis specifications for see v1.1, 2008.

[38] SOA4All Team. D11.2.1 initialise working group on aligning wsmo with other approaches to w3c v1.0, 2008.

[39] SOA4All Team. D1.2.1 wsmo grounding in sawsdl v1.0, 2008.

[40] Super Team. D6.3 process ontology reasoner, 2007.

[41] Gianluca Ripa Teodoro De Giorgio and Maurilio Zuccalà. An approach to enable replacement of soap services and rest services in lightweight processes. Master's thesis, CEFRIEL - ICT Institute Politecnico di Milano, 2010.

[42] Tomas Vitvar, Jacek Kopecký, Maciej Zaremba, and Dieter Fensel. Wsmo-lite: Lightweight semantic descriptions for services on the web. In *ECOWS*, pages 77–86. IEEE Computer Society, 2007.

[43] Tomas Vitvar, Jacek Kopecky, Jana Viskova, and Dieter Fensel. Wsmo-lite annotations for web services. In Manfred Hauswirth, Manolis Koubarakis, and Sean Bechhofer, editors, *Proceedings of the 5th European Semantic Web Conference*, LNCS, Berlin, June 2008. Springer Verlag.

[44] W3C/MIT. Hypertext transfer protocol – http/1.1. `http://tools.ietf.org/html/rfc2616`.

[45] W3C Semantic Web. Rdfa distiller and parser. `http://www.w3.org/2007/08/pyRdfa/`.

[46] Shang-Pin Ma Yu-Yen Peng and Jonathan Lee. Rest2soap: a framework to integrate soap services and restful services. In *Service-Oriented Computing and Applications SOCA (2009)*.

APPENDICE A

FILES XML

A.1 SAWSDL getForecast

```
<wsdl:definitions
...
xmlns:tns="http://it.crf.TheWeather/"
targetNamespace="http://it.crf.TheWeather/"
...
>
<wsdl:types>
 <xs:schema elementFormDefault="qualified"
     targetNamespace="http://it.crf.TheWeather/">
  <xs:element name="getForecast"
   sawsdl:modelReference="http://www.soa4all.eu/models/
          weatherForecasts#weatherForecastRequest">
   <xs:complexType>
        <xs:sequence>
         <xs:element minOccurs="1" maxOccurs="1" name="latitude" type="xs:double"
           sawsdl:modelReference="http://www.soa4all.eu/
                  models/GPS#latitude" />
         <xs:element minOccurs="1" maxOccurs="1" name="longitude" type="xs:double"
           sawsdl:modelReference="http://www.soa4all.eu/
                  models/GPS#longitude" />
         <xs:element minOccurs="1" maxOccurs="1" name="hour" type="xs:int"
           sawsdl:modelReference="http://www.soa4all.eu/
                  models/time#timeDistance_hours" />
         <xs:element minOccurs="1" maxOccurs="1" name="min" type="xs:int"
           sawsdl:modelReference="http://www.soa4all.eu/
                  models/time#timeDistance_minutes" />
        </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="getForecastResponse"
   sawsdl:modelReference="http://www.soa4all.eu/models/
          weatherForecasts#weatherForecastResponse">
```

```
    <xs:complexType>
     <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="1"
              name="getForecastResult" type="tns:Forecast"/>
         </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:complexType name="Forecast"
    sawsdl:modelReference="http://www.soa4all.eu/
            models/weatherForecasts#weatherForecast">
    <xs:sequence>
         <xs:element minOccurs="0" maxOccurs="1" name="utc" type="xs:string"
          sawsdl:modelReference="http://www.soa4all.eu/models/time#UTCTime" />
         <xs:element minOccurs="0" maxOccurs="1" name="wx" type="xs:string"
      sawsdl:modelReference="http://www.soa4all.eu/
              models/weatherForecasts#WeatherConditionsIconURL" />
         <xs:element minOccurs="1" maxOccurs="1" name="weather" type="xs:string"
      sawsdl:modelReference="http://www.soa4all.eu/
              models/weatherForecasts#WeatherConditionsShortDescription" />
         <xs:element minOccurs="0" maxOccurs="1" name="mslp" type="xs:string"
          sawsdl:modelReference="http://www.soa4all.eu/
                  models/weatherForecasts#Pressure_hPa" />
         <xs:element minOccurs="0" maxOccurs="1" name="temp" type="xs:string"
          sawsdl:modelReference="http://www.soa4all.eu/
                  models/weatherForecasts#Temperature_Centigrades" />
         <xs:element minOccurs="0" maxOccurs="1" name="relh" type="xs:string"
          sawsdl:modelReference="http://www.soa4all.eu/
                  models/weatherForecasts#Humidity_Percentage" />
         <xs:element minOccurs="0" maxOccurs="1" name="prcp" type="xs:string"
          sawsdl:modelReference="http://www.soa4all.eu/
                  models/weatherForecasts#Precipitation_mm" />
         <xs:element minOccurs="0" maxOccurs="1" name="wind" type="xs:string"
          sawsdl:modelReference="http://www.soa4all.eu/
                  models/weatherForecasts#Wind_Knots"/>
         </xs:sequence>
    </xs:complexType>
   </xs:schema>
</wsdl:types>
<wsdl:message name="getForecastSoapIn">
 <wsdl:part name="parameters" element="tns:getForecast" />
</wsdl:message>
<wsdl:message name="getForecastSoapOut">
 <wsdl:part name="parameters" element="tns:getForecastResponse" />
</wsdl:message>

<wsdl:portType name="TheWeatherCRFSoap">
 <wsdl:operation name="getForecast">
  <sawsdl:attrExtensions
   sawsdl:modelReference="http://www.soa4all.eu/models/
           weatherForecasts#WeatherForecastByTimeLocation" />
  <wsdl:input message="tns:getForecastSoapIn" />
  <wsdl:output message="tns:getForecastSoapOut" />
 </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="TheWeatherCRFSoap" type="tns:TheWeatherCRFSoap">
```

```
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
 <wsdl:operation name="getForecast" >
  <soap:operation soapAction="http://it.crf.TheWeather/getForecast"
        style="document" />
  <wsdl:input>
   <soap:body use="literal" />
  </wsdl:input>
  <wsdl:output>
   <soap:body use="literal" />
  </wsdl:output>
 </wsdl:operation>
</wsdl:binding>
<wsdl:service name="TheWeatherCRF">
 <wsdl:port name="TheWeatherCRFSoap" binding="tns:TheWeatherCRFSoap">
  <soap:address location="http://inrete.dyndns.info/TheWeather/TheWeatherCRF.asmx"/>
 </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

A.2 hREST con MicroWSMO getForecastByLocation

```
<div typeof="wsl:Service" about="#svc1"
 xmlns:hr="http://www.wsmo.org/ns/hrests#"
 xmlns:wsl="http://www.wsmo.org/ns/wsmo-lite#"
 xmlns:sawsdl="http://www.w3.org/ns/sawsdl#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
 <h1>
  <a href="servizio.html">
   <span property="rdfs:label">Weather Forecast</span>
  </a> service API
 </h1>
 This is a service of <a rel="sawsdl:modelReference"
     href="http://www.soa4all.eu/models/weatherForecasts">
      Weather Forecast</a>
 <div rel="wsl:hasOperation">
  <span typeof="wsl:Operation" about="#op1">
   <h2>Operation
        <code property="rdfs:label">
          getForecastByLocation
        </code></h2>
   <span property="hr:hasMethod">GET</span>
   <code property="hr:hasAddress" datatype="hr:URITemplate">
    <a rel="sawsdl:modelReference"
        href="http://www.soa4all.eu/models/operations/
              weather#WeatherForecastByTimeLocation">
        http://demo.cefriel.it/rest/previsioni/
             getForecastByLocation?{-join|&amp;|lat,lon,hour,min}
        </a>
   </code>
   <span rel="wsl:hasInputMessage">
    <span typeof="wsl:Message"><strong>Parameters:</strong>
         <code>
          (<a rel="sawsdl:modelReference"
            href="http://www.soa4all.eu/models/parameters/GPS#latitude">
```

```
                lat
            </a>)
            <span property="rdfs:label">lat</span>
           </code>
           (<a rel="sawsdl:modelReference"
             href="http://www.soa4all.eu/models/general/
                   parameters#required">
               Required
             </a>)
             <i>[<a rel="sawsdl:modelReference"
                  href="http://www.soa4all.eu/models/general/types#double">double</a>]
             </i>
                  - the latitude of the location<br />
          </span>
    </span>
    <span rel="wsl:hasInputMessage">
         <span typeof="wsl:Message">
          <code>
           (<a rel="sawsdl:modelReference"
           href="http://www.soa4all.eu/models/parameters/GPS#longitude">lon</a>)
       <span property="rdfs:label"> lon</span>
           </code>
          (<a rel="sawsdl:modelReference"
          href="http://www.soa4all.eu/models/general/parameters#required">Required</a>)
     <i>[<a rel="sawsdl:modelReference"
           href="http://www.soa4all.eu/models/general/types#double">double</a>]
          </i>
     - the longitude of the location<br />
    </span>
   </span>
   <span rel="wsl:hasInputMessage">
  <span typeof="wsl:Message">
   <code>
   (<a rel="sawsdl:modelReference"
     href="http://www.soa4all.eu/models/parameters/time#timeDistance_hours">hour</a>)
    <span property="rdfs:label">hour</span>
   </code>
   (<a rel="sawsdl:modelReference"
     href="http://www.soa4all.eu/models/general/parameters#optional">Optional</a>)
   <i>
    [<a rel="sawsdl:modelReference"
      href="http://www.soa4all.eu/models/general/types#integer">integer</a>]
   </i>
  </span>
 </span>
 <span rel="wsl:hasInputMessage">
  <span typeof="wsl:Message">
   <code>
    (<a rel="sawsdl:modelReference"
     href="http://www.soa4all.eu/models/parameters/time#timeDistance_minutes">min</a>)
    <span property="rdfs:label">min</span>
   </code>
   (<a rel="sawsdl:modelReference"
    href="http://www.soa4all.eu/models/general/parameters#optional">Optional</a>)
   <i>
   [<a rel="sawsdl:modelReference"
```

```
    href="http://www.soa4all.eu/models/general/types#integer">integer</a>]
  </i>
  - the time of the forecast
 </span>
</span>
<span rel="wsl:hasOutputMessage">
 <span typeof="wsl:Message">
  <strong>Output value:</strong>  weather and temperature of Location
   in  <a rel="sawsdl:modelReference"
        href="http://www.soa4all.eu/models/general/mediatype#text/xml">xml</a>
   format type.
  <a rel="sawsdl:modelReference"
     href="http://demo.cefriel.it/previsioni/OperationIDOutputSchema.xsd">
     Xml Schema
  </a>
 </span>
</span>
</div>
```

A.3 File xsd del servizio RESTful

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:sawsdl="http://www.w3.org/ns/sawsdl#">
  <xs:element name="previsioniMeteo"
   sawsdl:modelReference="http://www.soa4all.eu/models/
          weatherForecasts#weatherForecastResponse">
   <xs:complexType>
    <xs:sequence>
      <xs:element name="temperatura" type="xs:string" minOccurs="1" maxOccurs="1"
          sawsdl:modelReference="http://www.soa4all.eu/models/
                 weatherForecasts#Temperature_Centigrades"/>
      <xs:element name="previsione" type="xs:string"  minOccurs="1" maxOccurs="1"
          sawsdl:modelReference="http://www.soa4all.eu/models/
                 weatherForecasts#WeatherConditionsShortDescription"/>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Stampato il: 4 luglio 2010
creato con: LaTeX