

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

**CORSO DI LAUREA MAGISTRALE IN
INGEGNERIA INFORMATICA**

TESI DI LAUREA

in
Linguaggi e Modelli Computazionali M

tuProlog AS A SERVICE: INTELLIGENZA PER SCENARI IoT

CANDIDATO:
Andrea Muccioli

RELATORE:
Prof. Enrico Denti

CORRELATORI:
Dott. Ing. Roberta Calegari
Prof. Andrea Omicini

Anno Accademico 2015/2016
Sessione III

*Alla mia famiglia, che mi ha sempre sostenuto,
e a Martina, che mi ha dato la forza di arrivare fino in fondo*

Sommario

Sommario.....	1
1. Introduzione	3
2. LPaaS – Intelligenza distribuita nell’IoT	5
3. Analisi dei Requisiti	8
4. Architettura	11
4.1 Applicazione server	12
4.2 Applicazioni client	20
5. Tecnologie	22
5.1 Web Services	22
5.2 SOAP-WSDL Web Services.....	23
5.3 RESTful Web Services	26
5.4 Java EE e EJB	33
5.5 JSON Web Token	35
6. Scelte tecnologiche	40
6.1 Interfaccia utente e business logic	40
6.2 Rappresentazione dei dati	46
6.3 Sicurezza	47
6.4 Persistenza.....	49
6.5 Motore Prolog	50
7. Progetto.....	53
7.1 Applicazione lato server	53
7.2 Interfacce di accesso al servizio.....	54

7.3	Sicurezza	59
7.4	Persistenza.....	62
7.5	Motore Prolog e concorrenza.....	63
7.6	Caricamento dati	65
7.7	Applicazioni Client	67
8.	Implementazione: un caso concreto.....	68
8.1	Illustrazione dello scenario applicativo	68
8.2	Realizzazione hardware	70
8.3	Realizzazione Software.....	72
8.4	Collaudo e risultati	78
9.	Conclusioni	81
10.	Appendici	83
10.1	Configurazioni e dati per il caso d'uso.....	83
10.2	Dai dati generati alle liste di avvisi: un esempio.....	88
10.3	Guida all'installazione.....	90
11.	Bibliografia.....	98

1. Introduzione

Negli ultimi anni abbiamo assistito ad una crescita esponenziale dell'Internet of Things^[IOTG], in uno scenario globale in cui la possibilità di interconnessione tra device è andata via via espandendosi; da prerogativa dei soli computer, questa possibilità è stata estesa a notebook, tablet e smartphone, fino ad arrivare a dispositivi che fino a qualche anno fa sarebbero sembrati irrealistici: elettrodomestici, lampadine, dispositivi indossabili, spazzolini elettrici, semafori e tanti altri ancora^[IOTR]. Tutti questi oggetti d'uso comune, smettono così di essere entità disconnesse che devono essere accedute necessariamente in modo diretto e diventano (grazie alla pervasività globale delle reti IP-based) in grado di interagire remotamente tra di loro in modo ricco. Le potenzialità future sono innumerevoli, e non è difficile immaginare un futuro in cui questi “smart-object”^[SOBJ] non solo siano in grado di condividere tra di loro dati e informazioni, ma che possano anche accedere l'uno all'altro in modo automatico e senza l'intervento umano o registrare preferenze e abitudini degli utenti per modificare il proprio comportamento.

Questa tesi parte dunque da questo contesto generale e si concentra su LPaaS (Logic Programming as a Service)^[LPaaS], un modello per l'implementazione di dispositivi intelligenti sull'Internet of Things. Un device basato su LPaaS deve essere in grado di adattare il proprio comportamento tramite un core intelligente realizzato utilizzando la programmazione logica, e di rendere disponibili le proprie funzionalità sulla rete as-a-service, ovvero come servizi generali invocabili allo stesso modo da utenti o altri dispositivi.

Con l'obiettivo finale di realizzarne un prototipo funzionante e di dimostrarne le funzionalità, si inizierà dunque esaminando il modello generale di LPaaS con le relative problematiche e caratteristiche distillandone quindi una serie di requisiti necessari per le fasi successive.

A partire da questi verrà quindi elaborata una architettura generale per la realizzazione del prototipo, astraendo il più possibile dalle tecnologie da utilizzare

per implementarla. Solo dopo aver esaminato le tecnologie in questione che meglio si adattano al nostro caso e aver selezionato le più appropriate, queste verranno utilizzate per l'implementazione dell'architettura in un prototipo funzionante che rispetti il modello LPaaS.

Infine il prototipo realizzato verrà messo alla prova in uno scenario applicativo realistico appropriato al contesto per cui è stato pensato, in modo da dimostrarne le possibilità e verificarne la piena funzionalità.

2. LPaaS – Intelligenza distribuita nell'IoT

È indubbio che ad oggi l'IoT sia uno degli ambiti di ricerca e sviluppo più popolari e in costante crescita e questa popolarità è dovuta probabilmente non solo al successo del concetto stesso, ma anche ad una confluenza di fattori legati allo sviluppo tecnologico degli ultimi anni, che hanno reso estremamente fertile il terreno per la sua crescita.

Una di queste sopracitate componenti è ovviamente la progressiva miniaturizzazione dei dispositivi elettronici legata a doppio filo al continuo aumento delle prestazioni e alla parallela diminuzione dei costi e consumi degli stessi. Questo consente di equipaggiare una sempre più ampia gamma di oggetti e “cose” con capacità di calcolo ad un costo sempre minore. Questi miglioramenti hanno spalancato le porte a innumerevoli nuove possibilità permettendo a reti IoT di venire implementate in un numero sempre più alto e diversificato di scenari^[IOTS]: reti di sensori per il monitoraggio della salute personale, reti per domotica e Home Security, o ancora automazione all'interno di veicoli, gestione risorse e controllo del traffico all'interno di un ambiente smart city e molti altri ancora.

Nasce inoltre il bisogno, indipendentemente dallo scenario applicativo, di avere un certo livello di intelligenza a livello locale^[AMI1], che consenta di rendere una rete dinamica e flessibile a fronte di cambi di requisiti, preferenze e richieste. Queste reti devono inoltre essere in grado di produrre informazioni di alto livello per l'utente, piuttosto che una statica rete di sensori e attuatori che si limitano a produrre dati raw e ricevere comandi da eseguire.

La necessità è di avere un sistema context-aware, che sia in grado di estrapolare informazioni dal contesto locale, e usare queste informazioni per adattare il comportamento e il funzionamento della rete agendo su quello dei singoli device. Questi dati di contesto non sono solamente dati ambientali, ma dati d'uso, preferenze e abitudini degli utenti^[AMI2].

Una prima possibilità è di implementare l'intelligenza necessaria per questo tipo di operazione in modo centralizzato: un sistema collegato a tutti i device di una rete, che si occupi di raccogliere i vari dati di contesto e li elabori. Il frutto di queste elaborazioni può essere quindi usato per modificare e regolare la rete di conseguenza. Questo tipo di soluzione tuttavia presenta diversi problemi; sistemi centralizzati di questo tipo, specie per reti con un grande numero di nodi richiederebbero una grande potenza computazionale per scalare correttamente, inoltre è necessario che i singoli device possano collegarsi ad esso, cosa non banale, perché alcune applicazioni IoT potrebbero essere isolate da Internet.

La seconda possibilità si avvale dell'enorme progresso nelle performance e miniaturizzazione dei dispositivi di cui si è parlato. Ciò permette infatti di adottare una nuova visione sull'inserimento di intelligenza in una rete IoT, distribuendola sui singoli elementi che compongono la rete stessa. In questo modo le informazioni di contesto e, più in generale i dati, vengono elaborate all'interno del dispositivo stesso che così potrà fornire servizi e funzionalità più evolute, sia agli utenti sia agli altri dispositivi collegati.

L'evoluzione del comportamento della rete in questo modo avviene come somma dei cambiamenti sui singoli elementi che la compongono.

Il modo più semplice e naturale per implementare elementi che forniscano intelligenza all'interno di un sistema è quello di realizzarli utilizzando la programmazione logica^[L.BL]. Le tecniche di programmazione logica tradizionale tuttavia si scontrano con gli scenari tipici delle applicazioni IoT, dominati da modelli basati sulla mobilità degli elementi, largo utilizzo del cloud e funzionalità e risorse offerte as-a-service.

Per questi motivi nasce la necessità di declinare la programmazione logica classica nella prospettiva di uno scenario distribuito e orientato ai servizi come quello posto nell'IoT.

Logic Programming as a Service (LPaaS) rappresenta proprio questa evoluzione, e propone l'idea di usare i principi dei sistemi service-oriented per offrire sui singoli device funzionalità che normalmente sarebbero difficilmente utilizzabili in

remoto a causa delle limitazioni della programmazione logica, rendendole fruibili come servizi.

In questa nuova prospettiva si definisce un'interfaccia standard per accedere in modo controllato alle funzionalità di un motore inferenziale, in accordo ai principi SOA.

Si noti che questo modello è estremamente scalabile, e la sua applicazione non è limitata al mondo dell'IoT; potrebbe essere utilizzato, ad esempio, per realizzare un server centralizzato contenente al suo interno un motore inferenziale, le cui funzionalità siano offerte a multipli clienti tramite la stessa interfaccia e servizi standard. Ciò nonostante i vantaggi in uno scenario IoT sono notevoli; la realizzazione di un'interfaccia standard di accesso disaccoppia cliente e servitore, permettendo massima interoperabilità tra device anche molto eterogenei (cosa comune in IoT).

Il fatto che l'interfaccia di accesso sia standardizzata permette ai vari elementi della rete IoT di comunicare l'uno con l'altro senza avere informazioni specifiche sui propri interlocutori; è infatti il motore inferenziale interno a realizzare l'intelligenza vera e propria che determina il comportamento e le funzioni del device, l'interfaccia ne permette solamente l'accesso tramite canali predefiniti.

Nei capitoli successivi si procederà alla progettazione e realizzazione di un prototipo iniziale funzionante che realizzi il concetto di Logic Programming as a Service.

3. Analisi dei Requisiti

Lo scopo della tesi è dunque quello di realizzare un primo prototipo funzionante e il relativo caso d'uso, di un sistema server che segua il modello LPaaS, pensato perché ne venga fatto il deploy su device all'interno di una infrastruttura IoT, dove deve occuparsi di fornire intelligenza localizzata ed elaborazione di informazioni contestuali, permettendone inoltre l'accesso in remoto tramite un'interfaccia service-oriented.

Per prima cosa vengono analizzati i requisiti funzionali e non che il prototipo deve soddisfare:

- **Interfaccia di accesso web:** il server deve esporre una interfaccia di accesso per poter essere utilizzato in remoto da multipli client. Questa deve permettere di configurare ed effettuare il setup del servizio dinamicamente e di interrogare il motore d'inferenza. Dato che le funzionalità di configurazione e interrogazione sono logicamente separate e idealmente accedute da utenti diversi, risulta conveniente dividerle in due interfacce distinte.
 - **Interfaccia utente:** deve fornire servizi per richiedere la risoluzione dei goal contenuti in una lista di possibilità, oltre ad un servizio per consultare tale lista
 - **Interfaccia di configurazione:** deve essere accessibile solo a utenti autorizzati e fornire servizi per cambiare teoria, lista di goal possibili e configurazione.
- **Wrapping di un motore inferenziale:** il server deve contenere al suo interno un motore logico che realizzi la parte intelligente del sistema. Questo elemento non deve essere esposto direttamente all'interazione con i clienti ma deve essere accessibile solo in modo controllato tramite le interfacce utente.

- **Interazione con e senza stato di sessione:** deve essere possibile per il cliente interagire con il server sia in modo stateless che in modo stateful (conformemente a quanto previsto nel modello), per fare in modo che in richieste successive venga tenuto conto delle interazioni precedenti.
- **Sistema di autenticazione:** per far sì che l'interfaccia di configurazione sia accessibile solo ad utenti autorizzati, il server dovrebbe fornire un servizio di login o di autenticazione per limitare l'accesso a determinati servizi solo a utenti autorizzati.
- **Persistenza dei dati di configurazione:** i requisiti in termini di persistenza sono piuttosto limitati, ma deve essere comunque possibile garantire la persistenza della configurazione del server e di quella degli utenti e ruoli, in modo che sopravvivano ad uno shutdown del server.
- **Caricamento dei dati:** il server deve integrare un sistema per caricare all'interno del motore di inferenza dati generati dal server stesso o provenienti da sensori idealmente presenti sullo stesso dispositivo in cui esegue l'applicazione.
- **Disaccoppiamento:** il server deve essere completamente disaccoppiato dal client al quale non deve trasparire nulla della sua realizzazione interna. In modo analogo i servizi offerti dovrebbero essere utilizzabili da quante più tipologie di client diversi, indipendentemente dalla loro implementazione. Questa caratteristica è particolarmente importante in un ambiente con dispositivi estremamente eterogenei come avviene in IoT.
- **Light-weight:** il server deve essere pensato per funzionare con performance accettabili su dispositivi con risorse molto limitate, sia in termini di risorse computazionali, sia in termini di memoria a disposizione, ed essere comunque in grado di servire più clienti concorrentemente.
 - **Scalabilità aggiuntiva:** opzionalmente il server deve essere in grado di scalare anche su un numero molto elevato di utenti se portato su una macchina prestazionalmente superiore.

L'intelligenza localizzata di cui abbiamo parlato nel capitolo precedente (necessaria per la realizzazione del modello LPaaS), è dunque implementata da un motore inferenziale attorno al quale viene costruito un server leggero che si occupi di fornirgli i dati, regolarne l'accesso tramite interfacce accessibili in remoto e gestirne persistenza e sicurezza.

Ora dunque si può procedere elaborando un'architettura generale del sistema che realizzi e soddisfi tutti i requisiti illustrati in questo capitolo.

4. Architettura

Dai requisiti emergono alcuni elementi richiesti in modo abbastanza esplicito da permettere di modellare un'architettura generale da cui partire.

Si è deciso che l'applicazione lato server debba contenere al suo interno un motore inferenziale utilizzato per fornire l'intelligenza necessaria, calcolando le soluzioni richieste e per garantire context-awareness, venendo aggiornato con i dati ottenuti da sensori esterni o altre fonti. Il motore non deve essere accessibile in modo diretto dall'esterno e pertanto l'applicazione fornisce delle interfacce utente tramite le quali i client possono interagire con il motore stesso in modo controllato, configurandolo o interrogandolo. L'applicazione server deve occuparsi inoltre di rendere persistenti le impostazioni ricevute dall'utente configuratore affinché queste rimangano attive anche dopo un riavvio o lo shutdown del server, di validare e controllare l'accesso alle operazioni di configurazione affinché non siano disponibili senza le corrette credenziali, di gestire in modo concorrente le richieste provenienti da multipli client e di fornire mezzi per il mantenimento dello stato di sessione.

Per semplificare l'accesso al server può essere realizzata inoltre un'applicazione aggiuntiva opzionale che permetta di semplificare l'uso dei servizi astraendo dalla logica di comunicazione e che si occupi in modo trasparente di mantenere e gestire eventuali token o dati necessari per l'interazione con il server che devono essere mantenuti lato client.

A partire da queste informazioni possiamo elaborare una possibile architettura generale (Figura 4-1) dalla quale partire per poi scendere nei particolari specificandone meglio i dettagli e i singoli componenti.

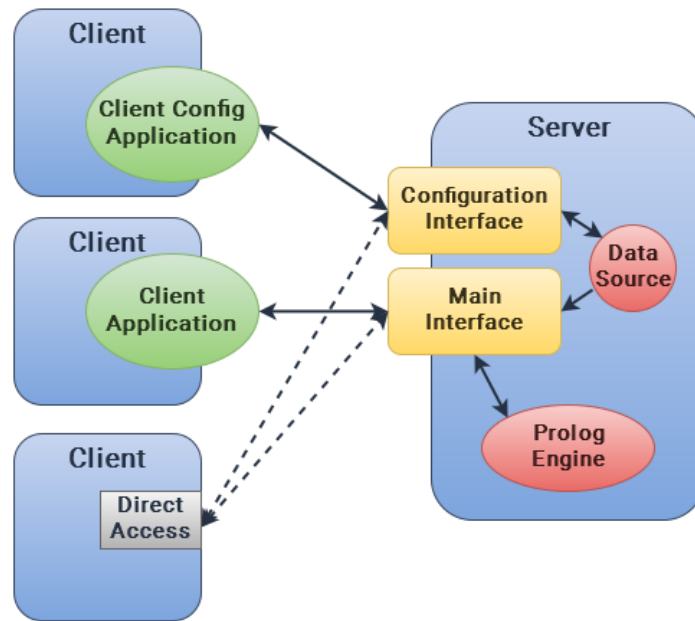


Figura 4-1- Schema dell'architettura generale del server da realizzare. Le interfacce consentono un accesso unificato e rappresentano l'unico modo di accedere al motore inferenziale. I client si possono collegare sia tramite applicazioni specifiche costruite ad hoc, sia in modo diretto.

4.1 Applicazione server

Per garantire l'estensibilità e manutenibilità dell'applicazione server, i suoi componenti possono essere distribuiti su più livelli d'astrazione. Così facendo si mantengono logicamente separati i componenti che realizzano funzionalità diverse all'interno dell'applicazione, permettendo l'interazione tra un livello e l'altro in modo controllato tramite metodi e chiamate precise, e garantendo la trasparenza dell'implementazione dei livelli sottostanti.

In Figura 4-2 viene data una possibile struttura lato server dell'applicazione che applichi i principi di separazione appena descritti e che rispetti i requisiti di disaccoppiamento forniti nel capitolo precedente. L'applicazione viene divisa in tre livelli funzionali seguendo un tipico pattern per architetture software^[3LA]. Il Presentation Layer qui è sostituito da un Interface Layer, che più che occuparsi di fornire una interfaccia grafica all'utente, si occupa di fornirne una logica. In modo del tutto simile però, queste non accedono mai direttamente al database, ma invocano solo operazioni sui componenti del Business Logic Layer, la cui implementazione resta comunque trasparente. Allo stesso modo la struttura interna

del server è completamente nascosta al client che accede all'applicazione solo tramite le interfacce fornite.

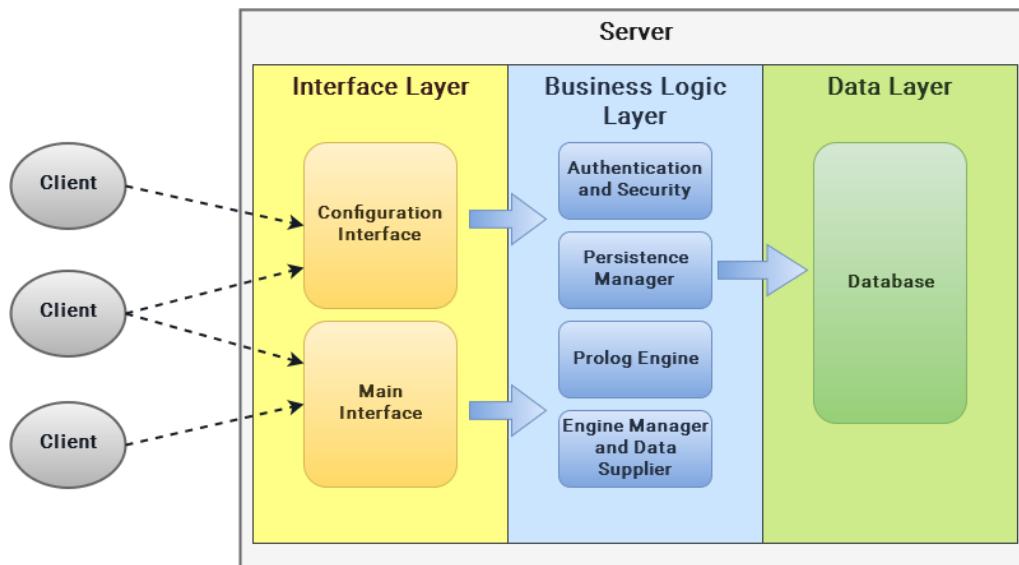


Figura 4-2- Divisione logica dell'applicazione server in layer d'astrazione separati ed interagenti.

L'applicazione, come da requisiti, può essere acceduta tramite due interfacce che rappresentano la vista del server da parte dei client. La logica di business è realizzata attorno all'engine Prolog interno all'applicazione e si occupa ad esempio di gestirne gli accessi e l'inserimento dei dati. La logica di persistenza si occupa dell'accesso al database, e della gestione dei dati su di esso, recuperarli quando richiesto e aggiornarli quando viene inoltrato un cambiamento della configurazione.

4.1.1 Interfacce di accesso al servizio

Come da requisiti l'utente può accedere al servizio tramite due differenti interfacce, una senza alcuna restrizione di accesso, che permette di svolgere una serie limitata e prestabilita di operazioni, anche sul motore Prolog interno, e un'altra ad accesso limitato, con richiesta di credenziali, che permette di gestire la configurazione del servizio per i normali utenti e in particolare il comportamento e le operazioni permesse sul motore interno.

La separazione delle due interfacce è logica, entrambe insistono infatti sugli stessi componenti del Business Logic Layer, ma in questo modo si effettua una divisione in base alle funzionalità offerte all'utente; ad esempio ad un utente normale

potrebbe essere fornita solo l'interfaccia principale d'accesso, lasciandolo all'oscuro dell'esistenza stessa di una interfaccia di configurazione.

Entrambe le interfacce offrono un'operazione `getInfo()` che restituisce, generandola a runtime, una rappresentazione dell'interfaccia corrispondente, contenente le operazioni disponibili, i rispettivi indirizzi, il tipo degli argomenti necessari e dei risultati restituiti e la necessità o meno di autenticazione.

Questa operazione non è strettamente necessaria, tuttavia è utile per fornire una descrizione completa dei servizi al cliente, che può quindi utilizzarla per scoprire dinamicamente le varie funzionalità offerte a partire proprio dal risultato della `getInfo()`.

L'interfaccia di configurazione deve essere ad accesso limitato e di conseguenza permette l'uso di due sole operazioni senza essersi autenticati:

Metodo	Effetto
<code>getInfo()</code>	Genera una descrizione dei servizi offerti.
<code>login(user, psw)</code>	Il metodo che, date le credenziali, consente di autenticarsi e quindi accedere alle rimanenti operazioni dell'interfaccia di configurazione.

Tabella 4-1- Metodi dell'interfaccia di configurazione che non richiedono una precedente autenticazione

Ogni rimanente operazione ha come preconditione l'avvenuto login con successo.

Metodo	Effetto
<code>getTheory()</code> <code>setTheory(theory)</code>	Metodi per ottenere e settare la teoria del motore interno prolog del servizio.
<code>getGoalsList()</code> <code>setGoalsList(goalsList)</code> <code>addGoal(goal)</code> <code>getGoalByName(goal)</code> <code>removeGoal(goal)</code> <code>getGoalAtIndex(i)</code> <code>removeGoalAtIndex(i)</code>	Metodi per gestire la <code>goalsList</code> del servizio. L'utente configuratore è responsabile della creazione e impostazione di una lista di possibili goal di cui richiedere la risoluzione al motore Prolog del servizio. I normali utenti potranno richiedere la risoluzione di un goal al servizio esclusivamente se questo è contenuto all'interno della <code>goalsList</code> configurata in quel momento.
<code>getConfiguration()</code> <code>setConfiguration(config)</code>	Metodi per ottenere e settare ulteriori configurazioni per il servizio. Questi metodi sono presenti per possibili usi futuri e attualmente non hanno alcun effetto.
<code>reset()</code>	Metodo per resettare la configurazione esistente.

Tabella 4-2- Metodi dell'interfaccia di configurazione che richiedono un avvenuto login con successo

Tutte le modifiche effettuate e le impostazioni configurate devono essere salvate all'interno dal server; inoltre, per far sì che sopravvivano allo spegnimento o al riavvio del server stesso, è auspicabile che vengano salvate in memoria persistente (si veda il successivo capitolo: Persistenza)

L'interfaccia principale di accesso al servizio deve contenere metodi per controllare la configurazione del servizio attualmente impostata (da una precedente configurazione tramite l'interfaccia di cui sopra), e per richiedere soluzioni di determinati goal al motore d'inferenza.

La risoluzione dei goal deve poter avvenire in due modalità differenti: con sessione o senza sessione; la risoluzione senza sessione implica che la chiamata stessa sia auto contenuta, successive chiamate richiederanno di specificare nuovamente il goal, e anche in caso di richieste successive identiche, il motore fornirà (se lo stato interno non è cambiato) risposte uguali. Queste operazioni, a parità di configurazione e stato del motore Prolog, sono dunque idempotenti. La risoluzione con sessione invece deve mantenere, appunto, uno stato di sessione, e successive invocazioni con lo stesso goal forniranno le soluzioni successive a partire dall'ultima ottenuta.

In particolare le operazioni offerte dall'interfaccia principale sono:

Metodo	Effetto
getInfo()	Genera una descrizione dei servizi offerti.
getTheory()	Restituisce la teoria correntemente impostata nel motore Prolog interno e che verrà utilizzata per la risoluzione dei goal.
getGoalsList()	Restituisce la lista di goal correntemente impostata, e che contiene gli unici goal che l'utente può selezionare per la risoluzione.
getGoalByName(goal)	Metodo che reindirizza al goal della goals list corrispondente a quello selezionato.
getGoalAtIndex(i)	Restituisce il goal all'indice i nella goalsList.

getSolution(goal) getSolutions(goal, n)	Metodi per richiedere la soluzione (o le prime n soluzioni) di un goal che deve necessariamente appartenere alla goalsList. Il parametro n di getSolutions è opzionale, in caso sia assente, vengono richieste tutte le soluzioni disponibili. Queste operazioni non forniscono stato di sessione.
getSolutionWithSession(goal) getSolutionsWithSession(goal, n)	Metodi equivalenti a quelli precedenti, ma con supporto allo stato di sessione.

Tabella 4-3- Metodi dell'interfaccia utente principale

Per fornire una interazione stateful, come stabilito a livello di requisiti, devono essere fornite anche operazioni apposite che consentano di gestire il derivante stato di sessione. Il problema in realtà non è banale sia per eventuali problemi di concorrenza, sia per motivi legati alle specifiche tecnologie usate per implementare l'applicazione stessa. La sessione può essere mantenuta lato client, lato server, mantenuta in memoria persistente e recuperata tramite un ID, ecc... Per questi motivi si rimanda il discorso a capitoli successivi (in particolare per maggiori dettagli sul mantenimento della sessione si vedano i capitoli: il capitolo 4.1.4 e 7.2).

Inoltre per evitare di appesantire inutilmente il servizio, è preferibile non abusare delle operazioni con sessione quando non necessarie, e piuttosto utilizzare una nuova sessione o le operazioni senza sessione quando si deve richiedere la risoluzione di un nuovo goal rispetto alla sessione già aperta.

4.1.2 Sicurezza

Sebbene l'accesso al motore Prolog da parte dei client non avvenga mai in modo diretto, ma solamente attraverso la rispettiva interfaccia di accesso, non ci sono limitazioni ai goal inseribili nella goalsList. Questo, di fatto, consentirebbe potenzialmente l'esecuzione di qualsiasi comando da parte dei client, previo inserimento nella goalsList.

Per questa ragione è importante che il configuratore del servizio inserisca solamente goal adeguati; in particolare è sconsigliabile inserire nella goalsList

comandi che modifichino la knowledge base dell'engine Prolog o il suo stato (questo aspetto viene approfondito in seguito nel capitolo 4.1.4).

Anche per questo è importante che le operazioni di configurazione siano accessibili solo ai client che posseggano le dovute credenziali d'accesso.

Si possono definire quindi tre ruoli principali:

- **user/guest:** possibilità di accedere al servizio tramite l'apposita interfaccia utente. Può richiedere la risoluzione di goal contenuti nella goalsList precedentemente configurata. Non necessita di credenziali.
- **configurator:** possibilità di modificare le modalità di accesso al motore Prolog tramite l'apposita interfaccia, oltre ai permessi precedenti. **Richiede credenziali valide.**
- **admin:** accesso illimitato all'applicazione. Possibilità di modificare la configurazione e il funzionamento del server, le credenziali e i ruoli degli utenti, oltre a tutti i permessi precedenti. **Richiede credenziali valide.**

Inoltre è importante che stato e informazioni inviati dal server al client per poi essere rinviati indietro al server quando richiesti, non vengano in alcun modo modificati dal client stesso o da terzi e auspicabilmente non siano nemmeno in chiaro. È quindi necessario che le informazioni inviate vengano firmate e crittate per avere la certezza della loro provenienza e della loro integrità quando vengono inviate indietro dal client.

4.1.3 Persistenza

Come già visto le esigenze dell'applicazione server in termini di persistenza sono piuttosto limitate. L'applicazione infatti si appoggia su un database solamente per salvare le impostazioni ricevute tramite l'interfaccia di configurazione e le informazioni relative agli utenti: username, password, e ruolo assegnato, necessario per effettuare la validazione al momento del login.

Quindi, sebbene sia potenzialmente possibile spostare il livello di persistenza su un database esterno, in questo particolare caso, considerato quanto detto sopra e il

fatto che l'obiettivo finale è la realizzazione di un server leggero e auto-contenuto, risulta conveniente mantenere unite l'applicazione server e il livello di persistenza. Dal punto di vista architetturale molti dei moduli del Business Logic Layer e Interface Layer necessitano di accedere ai dati persistiti su database (ad esempio le interfacce e il motore inferenziale per accedere alla configurazione del servizio, il modulo di sicurezza per i dati sugli utenti). Questi accessi non avvengono in modo diretto (le interfacce tra l'altro, dato il pattern a 3 layer scelto per l'architettura, non hanno visione del Data Layer), ma devono passare necessariamente per il modulo "Persistence Manager". Questo elemento ha un ruolo di interfaccia tra Data Layer e Business Logic Layer (in modo analogo a quanto fatto dalle interfacce utente per l'accesso al server), con tutti i vantaggi annessi, quali una migliore separazione dei layer, l'accesso trasparente al database da parte di tutti i moduli, e in generale una gestione centralizzata della persistenza.

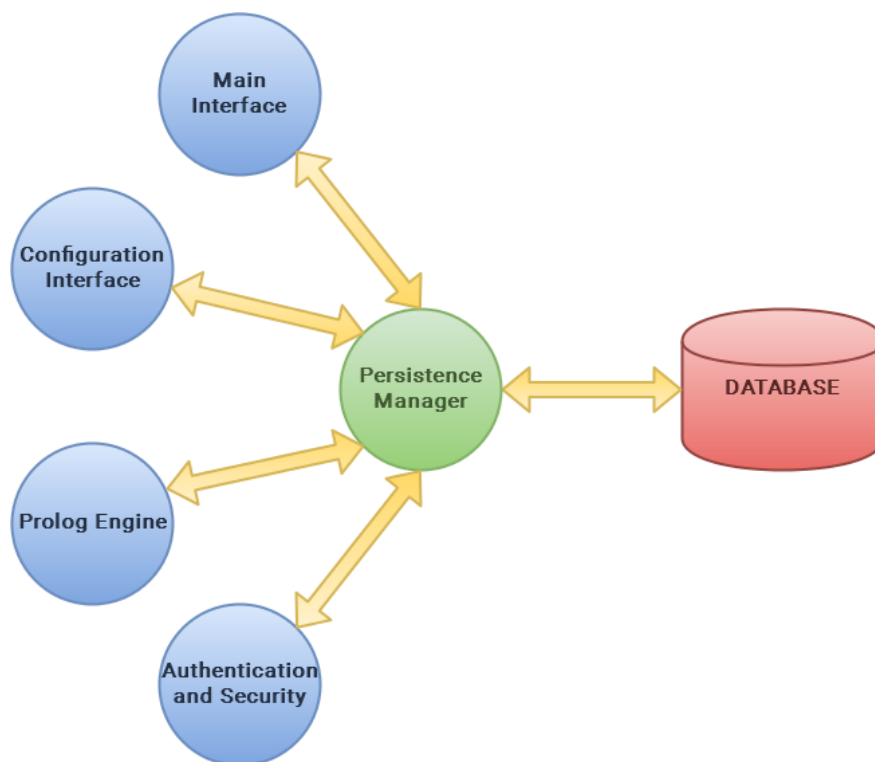


Figura 4-3- L'accesso al database da parte di tutti i moduli dell'applicazione non avviene in modo diretto ma attraverso i metodi offerti da Persistence Manager

4.1.4 Motore inferenziale e concorrenza

Al cuore della logica di business del server c'è il motore inferenziale che viene utilizzato dall'applicazione per la risoluzione dei goal richiesti tramite interfaccia utente. Le soluzioni ottenute vengono rimandate quindi al client come risposta.

La knowledge base dell'engine e in generale il suo stato pongono tuttavia problemi di concorrenza. Infatti nel momento in cui ai client (che per la natura stessa dell'applicazione, vi accedono in modo concorrente) viene permesso di chiedere la soluzione di goal che modificano lo stato dell'engine o la sua knowledge base (basti pensare ad operazioni di caricamento di librerie, assert o retract di termini) nascono immediatamente problemi di coerenza dell'engine, in particolare in caso di invocazioni con sessione in cui il motore Prolog può trovarsi in stati diversi in differenti sessioni cliente.

Escludendo la possibilità estrema, ma chiaramente inaccettabile per un'applicazione server, di eliminare ogni concorrenza e serializzare tutte le chiamate, una soluzione meno drastica può essere quella di considerare un motore “quasi” statico; evitando di inserire goal la cui soluzione modifica lo stato o la knowledge base dell'engine nella goalsList del servizio, e limitando questa possibilità alla logica interna del server.

Questo avviene in occasioni ben definite, ad esempio alla ricezione di nuovi dati da un sensore. In questo caso il server aggiorna lo stato dell'engine e invalida le sessioni in corso. Questa tecnica risulta molto più complicata da applicare non ponendo restrizioni sui goal accettabili; sarebbe necessario dividere tra operazioni che non modificano lo stato e operazioni che lo fanno, e invalidare tutte le sessioni ogni volta che una di queste ultime operazioni viene richiesta.

Utilizzando invece la soluzione dell'engine “quasi” statico, risulta persino possibile dividere l'engine in due parti, una singola e condivisa, che ne contenga lo stato aggiornato di volta in volta, e la seconda costituita da numerose istanze equivalenti di motori Prolog creati a partire dallo stato unico. Queste istanze multiple possono così servire concorrentemente più richieste contemporanee.

In questo modo si ha una evoluzione dell'engine, che non risulta totalmente statico e ottiene inoltre i vantaggi prestazionali di una esecuzione concorrente delle richieste. Il costo, d'altra parte, è la perdita di parte delle potenzialità offerte dal linguaggio.

4.2 Applicazioni client

Per semplicità e simmetria con le due interfacce utente lato server, anche l'applicazione client è divisa in due “sotto-applicazioni” una per l'interazione con l'interfaccia principale del servizio, e l'altra per l'interazione con l'interfaccia di configurazione.

I metodi della classe di configurazione ricalcano in modo abbastanza preciso le operazioni disponibili nell'interfaccia corrispondente lato server, tuttavia consentono di accedervi gestendo le procedure di login, di autenticazione ad ogni chiamata, e tutta la logica per le chiamate remote in modo quanto più trasparente possibile.

Metodo	Effetto
getInfo()	Restituisce le informazioni sull'interfaccia di configurazione
login(user, psw)	Effettua il login utilizzando le credenziali specificate
getTheory() setTheory(theory)	Metodi per ottenere e settare la teoria del servizio
setGoals(goalsList) getGoals addGoal(goal) removeGoal(goal)	Metodi per gestire la lista di goal resi disponibili dal servizio
reset()	Metodo per resettare la configurazione attualmente impostata

Tabella 4-4- Metodi dell'applicazione client di configurazione

Anche la classe per accedere all'interfaccia principale fornisce prevalentemente metodi che ricalcano le operazioni offerte dal server, con l'aggiunta però di metodi per selezionare il goal da risolvere e per chiudere una sessione correntemente aperta.

Metodo	Effetto
getInfo()	Restituisce le informazioni sull'interfaccia principale
getTheory()	Metodo per ottenere la teoria correntemente impostata.
getGoalsList()	Metodo per ottenere la lista di goal attualmente disponibili.
getGoal() setGoal(goal)	Metodo per selezionare il goal che verrà risolto nelle successive chiamate. Il goal deve appartenere alla goalsList.
getSolution() getSolutions(n) getSolutions()	Metodi per richiedere una, n o tutte le soluzioni per il goal selezionato al server. Questi metodi non utilizzano lo stato di sessione.
getSolutionWithSession() getSolutionsWithSession(n) getSolutionsWithSession()	Metodi per richiedere una, n o tutte le soluzioni per il goal selezionato al server. Questi metodi utilizzano lo stato di sessione.
closeSession()	Chiude la sessione corrente. La successiva chiamata con sessione creerà una nuova sessione.

Tabella 4-5- Metodi dell'applicazione client di accesso principale

5. Tecnologie

In questo capitolo viene fatta una veloce panoramica di alcune tecnologie fortemente rilevanti per la comprensione di questa tesi, al fine di sintetizzare le loro caratteristiche principali.

5.1 Web Services

Un Web Service^[WS] è un sistema software progettato per supportare l'interazione tra due dispositivi attraverso la rete, utilizzando un sistema a scambio di messaggi. Una funzione software viene così resa disponibile per l'invocazione remota, rendendola pubblica attraverso un indirizzo di rete ad essa associato.

Un cliente che vuole invocare un particolare servizio, dopo averlo conosciuto, deve preparare un messaggio contenente le informazioni necessarie all'invocazione per poi inviarlo al provider del servizio, dove avviene l'esecuzione vera e propria e, se prevista, viene generata una risposta (sempre tramite messaggio).

Questo sistema a scambio di messaggi è volto soprattutto a disaccoppiare cliente e servitore dal punto di vista del linguaggio di implementazione. Il Web Service può essere realizzato in qualsiasi linguaggio fintanto che sia possibile ricevere e interpretare messaggi, e la sua implementazione resti trasparente ai clienti. Al fine di realizzare ciò, cliente e servitore devono poter comunicare utilizzando messaggi scritti in un linguaggio comune e interpretabile praticamente da qualsiasi software (XML).

Questo tipo di sistema per altro garantisce un alto livello di componibilità. Più Web Service possono interagire tra loro, chiedendo i rispettivi servizi, per andare a formare reti anche di grandi dimensioni e svolgere operazioni complesse.

Sebbene i Web Service siano stati inizialmente pensati per utilizzare un particolare set di tecnologie e linguaggi (SOAP, WSDL, UDDI), il loro successo ha portato ad altre implementazioni, con altre tecnologie, dello stesso concetto di base.

5.2 SOAP-WSDL Web Services

Questa è l'implementazione iniziale di Web Service, e quella che utilizza le tecnologie inizialmente previste dallo standard.

5.2.1 *Caratteristiche*

In questo modello il servizio è descritto da un file in formato WSDL (Web Services Description Language), un linguaggio formale in formato XML, che ne descrive ogni operazione disponibile e le modalità di comunicazione. Il file WSDL rappresenta un contratto tra server e client e fornisce informazioni su:

- Quali servizi sono invocabili dal client.
- Il protocollo di comunicazione da usare per l'invocazione di suddette operazioni (formato dei messaggi di request e dei messaggi di response).
- L'endpoint del servizio che, solitamente, corrisponde all'URI alla quale è reso disponibile il Web Service.

Il WS viene reso disponibile a terzi pubblicandone l'interfaccia su un servizio di directorio chiamato UDDI (Universal Description, Discovery and Integration). Questo fornisce informazioni su tutti i WS su di esso pubblicati, viene interrogato dai client e fornisce loro il collegamento ai file WSDL dei servizi richiesti.

Le interazioni avvengono tutte tramite messaggi SOAP (Simple Object Access Protocol), sia quelle tra cliente e UDDI sia quelle tra cliente e servitore. SOAP è un linguaggio basato su XML, con cui avviene lo scambio dei dati. I messaggi SOAP inviati e ricevuti da server e client durante l'invocazione di un Web Service, vengono validati utilizzando le informazioni sul formato dei messaggi, contenute nel file WSDL.

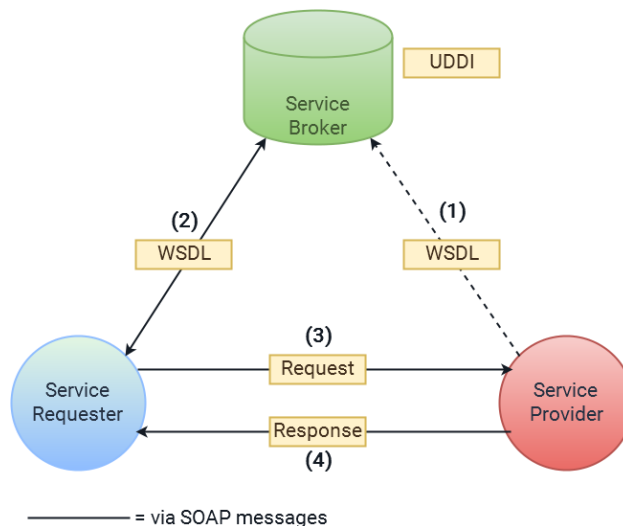


Figura 5-1- Schema di comunicazione in WS. Il SP registra un proprio servizio su un SB passandogli il WSDL corrispondente (1). Il SR richiede un servizio al SB che risponde con il WSDL del servizio richiesto (2). Utilizzando le informazioni nel file WSDL SR invia una richiesta di invocazione del servizio al SP (3) che esegue e risponde con i risultati (4).

Il protocollo SOAP può poggiare su diversi protocolli di trasporto sottostanti, tuttavia HTTP è il più comunemente utilizzato ed è l'unico ad essere stato standardizzato dal W3C.

5.2.2 Analisi comparativa

Vantaggi:

- **Disaccoppiamento:** il sistema a scambio di messaggi su cui si basa WS permette di disaccoppiare architettura e linguaggio con cui è realizzato il servizio da quelli dei suoi client, permettendo grande interoperabilità tra sistemi diversi.
- **Alta disponibilità:** i WS vengono pubblicati tramite internet, e sono pertanto disponibili da chiunque possieda una connessione internet. Il client può inoltre scegliere tra diversi servizi realizzati in modo diverso, per ottenere quello che più si addice alle sue necessità.
- **Riusabilità:** è possibile rendere disponibile codice non inizialmente pensato per essere pubblicato o codice legacy come WS con uno sforzo minimo.
- **Standard:** questo tipo di Web Service è costruito attorno ad uno standard molto ben strutturato, con linguaggi specifici e che si presta bene

all'introduzione di funzionalità più avanzate tramite estensioni (WS-Security, WS-Addressing, WS-AtomicTransaction, ecc...).

Svantaggi:

- **Complicatezza:** sebbene i WS siano molto ben strutturati, questa struttura rende spesso la creazione, pubblicazione e uso di WS più complicata del dovuto, anche nella loro versione base, senza alcuna estensione.
- **Pesantezza:** molto overhead nella comunicazione: ottenimento del file WSDL, creazione del canale, preparazione del messaggio SOAP, invio, validazione del messaggio SOAP di risposta tramite WSDL, parsing XML del messaggio.
- **Inefficienza:** le informazioni vengono scambiate esclusivamente in messaggi di tipo testuale. Questo comporta sia problemi riguardo alla necessità di rappresentare i dati scambiati esclusivamente tramite plain text, sia per il fatto che lo scambio di dati in formato testuale è intrinsecamente meno efficiente di quello in formato binario.
- **Statelessness:** sebbene non necessariamente un difetto, può diventare complicato rappresentare interazioni intrinsecamente stateful utilizzando questo sistema. In realtà sono disponibili opzioni nelle estensioni. A tal riguardo si veda il capitolo successivo.

5.2.3 WS-Addressing e statefulness

WS-Addressing^[WS-A] è un'estensione per WS che consente di aggiungere informazioni di routing all'header del messaggio SOAP. Un messaggio che utilizzi WS-Addressing può contenere informazioni specifiche su come essere consegnato ad un particolare end-point, piuttosto che fare affidamento al sistema di routing del livello di trasporto sottostante, che a questo punto diventa responsabile solo dell'affidamento del messaggio ad un dispatcher, specificato tramite URI, in grado di interpretare le informazioni WS-Addressing.

Questa estensione in realtà permette anche di superare l'intrinseca statelessness dell'architettura WS. Vengono introdotti infatti campi come "ws:replyTo"

all'header del messaggio SOAP, contenente l'EPR (EndPoint Reference) al quale la risposta deve essere inviata. In questo modo è possibile creare una sessione duratura tra questo endpoint e il server, disaccoppiando il tempo di vita dell'interazione SOAP da quello delle singole interazioni HTTP sottostanti.

WS-Addressing tuttavia è supportato in modo molto disomogeneo dai vari application server e la possibilità di creare WS stateful non è molto utilizzata, sia perché va in parte contro al concetto stesso di leggerezza e statelessness su cui si basano i WS, sia perché questa necessità è stata in parte soddisfatta in modo alternativo e più elegante dai sempre più diffusi RESTful Web Services.

5.3 RESTful Web Services

Nel momento in cui si decide di poggiare l'architettura, almeno in teoria generica dei Web Services solo su HTTP, gran parte della struttura che garantiva questa generalizzazione non è più necessaria, e diventa possibile semplificare di molto il modello. Questo è quanto avviene con i RESTful Web Services, un'implementazione dei Web Services che utilizza HTTP (sebbene non obbligatorio da specifica, è comunque lo standard de-facto) per l'invio e ricezione dei messaggi, per l'accesso alle risorse e che si basa sullo stile architetturale REST.

5.3.1 REST

REST (REpresentational State Transfer)^[REST] è uno stile architetturale che definisce una serie di vincoli di progettazione per applicazioni Web con l'obiettivo di raggiungere una serie di caratteristiche desiderate (scalability, simplicity, modifiability, visibility, portability e reliability). Questi vincoli sono:

- **Client-Server:** La separazione di compiti e ruoli tra client e server permette di aumentare la portabilità dell'interfaccia utente e contemporaneamente migliorare la scalabilità semplificando le componenti server. Inoltre, ancora più importante, il server e il client possono evolvere, cambiare la propria implementazione, essere sostituiti, indipendentemente l'uno dall'altro, fintanto che l'interfaccia tra i due non viene cambiata.

- **Stateless:** Ogni richiesta da client a server deve essere stateless (stato di sessione) e contenere tutte le informazioni necessarie a interpretarla correttamente; a tal fine non devono essere utilizzate informazioni salvate sul server. Lo stato di sessione deve essere quindi mantenuto interamente sul client. Il server stesso può tuttavia avere un suo stato interno (non parliamo più di session state), dato ad esempio dallo stato delle risorse che offre in un particolare istante; particolari richieste del client portano il server ad un cambiamento di questo stato. L'intera interazione con i client porta quindi all'evoluzione dello stato del server nel tempo.
- **Cacheable:** per migliorare l'efficienza delle comunicazioni, si richiede che i dati contenuti in una risposta ad una determinata richiesta, vengano esplicitamente etichettati come cacheable o non-cacheable. In caso positivo al client è concesso mantenere in cache quella risposta, per essere riusata in seguito a fronte di una richiesta equivalente.
- **Layered System:** si limita la visibilità dei componenti ad un solo livello, in questo modo un client non può sapere se è connesso ad un intermediario o destinatario finale della richiesta. Oltre ad aumentare la scalabilità del sistema, il sistema a livelli può essere utilizzato per incapsulare servizi legacy o per proteggere nuovi servizi da clienti legacy.
- **Code on demand (opzionale):** il server può temporaneamente estendere o incrementare le funzionalità di un client, trasferendo codice eseguibile in forma di componenti precompilati (come applet Java) o di script eseguibili (come JavaScript).
- **Uniform Interface:** un'interfaccia uniforme di accesso è fondamentale per il disaccoppiamento tra client e server, in modo che porti ad una totale indipendenza l'uno dall'altro come discusso precedentemente. Per realizzare questo punto ulteriori quattro vincoli sono necessari:
 - **Identificazione delle risorse:** le risorse devono essere univocamente associate ad un identificatore (URI), utilizzato dal client per indirizzare richieste ad una specifica risorsa.

- **Manipolazione delle risorse tramite le loro rappresentazioni:** le risorse vere e proprie sono concettualmente separate dalla loro rappresentazione. Il client riceve quest'ultima, e non la risorsa stessa. Inoltre un server può fornire ai client diverse rappresentazioni di una stessa risorsa (ad esempio una in XML, una in JSON, una in CSV, ecc...), e tutte queste possono essere ulteriormente diverse dalla sua rappresentazione interna. La rappresentazione di una risorsa deve contenerla completamente. Se è necessario rappresentare parzialmente una risorsa, allora si dovrebbe considerare di dividerla in sottomrisorse.
- **Self-descriptive messages:** ogni messaggio deve contenere abbastanza informazioni e metadati per descrivere come il messaggio vada processato. Ad esempio specificandone il formato del contenuto.
- **Hypermedia As The Engine Of Application State (HATEOAS):** un'applicazione REST deve comportarsi in modo simile ad un ipertesto o un sito Web, fornendo dinamicamente al client link ad altre risorse o alle azioni eseguibili su di esse, in modo che il client possa “navigare” in modo dinamico l'applicazione e le sue risorse.

5.3.2 Service Oriented Architecture e Resource Oriented Architecture

Se si cerca di implementare uno stesso gruppo di funzionalità come Web Services, si ottengono interfacce di accesso a queste funzionalità ben diverse, a seconda che si usi un approccio classico (SOA) o uno RESTful (ROA).

Nei Web Services classici di tipo SOAP-WSDL i servizi sono realizzati in modo simile a metodi remoti, possono essere invocati remotamente ed eseguono una particolare funzione, solitamente descritta dall'URL alla quale sono raggiungibili. Ad esempio potrei avere un'applicazione che mappa come segue in suoi metodi in servizi:

- `getResource(id)`; mappato all'URL: `http://myapp.net/getResource`

- `setResource(id, r)`; mappato all'URL: `http://myapp.net/setResource`
- `deleteResource(id)`; mappato all'URL: `http://myapp.net/deleteResource`

In ottica ROA questa implementazione è sbagliata (sarebbe realizzabile comunque, ma andrebbe contro i principi precedentemente discussi), poiché deve essere data, invece, centralità alle risorse. Seguendo l'esempio precedente, avremmo la risorsa univocamente identificata dall'URL: `http://myapp.net/resources/1234`, supponendo che 1234 sia l'id ad essa associato. Le diverse operazioni avvengono inviando richieste differenti a quello stesso indirizzo (in HTTP queste operazioni sono rappresentate dai diversi “verb” del protocollo HTTP, si veda il capitolo successivo a tal riguardo); inviando quindi una richiesta “get” a quell'indirizzo, riceverei la rappresentazione della risorsa, inviando una richiesta “set” e specificando nel corpo della richiesta una nuova rappresentazione della risorsa, il server cambierebbe la sua rappresentazione interna per riflettere quella ricevuta. Infine, inviando una richiesta “delete” il server rimuoverebbe quella risorsa, e magari reindirizzerebbe l'utente ad una lista di tutte le risorse rimaste, all'indirizzo `http://myapp.net/resources/`.

5.3.3 REST applicato ai Web Services

A differenza dei Web Services tradizionali basati su SOAP-WSDL, non c'è uno standard ufficiale per l'implementazione di WS RESTful, dato che REST è uno stile architetturale secondo il quale viene declinato il concetto di Web Services, e non un protocollo vero e proprio come può invece essere SOAP. Riprendendo l'esempio precedente, nulla impedisce di implementare un RESTful Web Service con l'interfaccia e la struttura di un WS tradizionale, tuttavia questo va contro il senso stesso di utilizzare REST.

RESTful Web Services poggiano sull'infrastruttura Web esistente e utilizzano HTTP per la comunicazione; pertanto per identificare le risorse vengono utilizzate URL, per la rappresentazione delle risorse vengono utilizzati formati standard e largamente utilizzati e riconosciuti, specie in ambito Web, come JSON^[JSON] e XML, e si utilizzano i tipi MIME^[MIME] per specificare il formato dei dati attesi in una

richiesta o da aspettarsi in una risposta del Web Service. I messaggi con i quali client e server comunicano sono normalissimi messaggi HTTP, senza utilizzo di protocolli sovrastanti come SOAP.

HTTP Request:

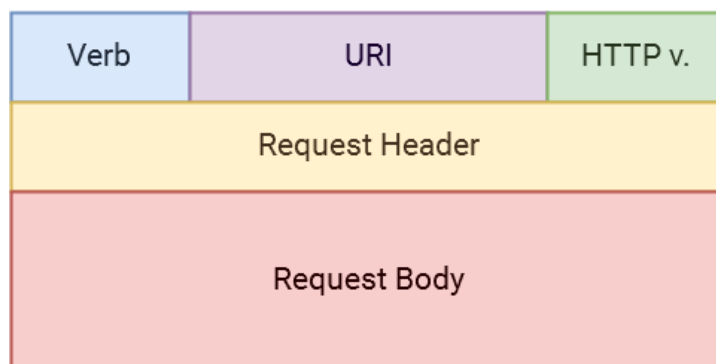


Figura 5-2- Struttura di una richiesta HTTP

In RESTful WS avviene una cosa abbastanza peculiare e si utilizzano le particolarità del protocollo HTTP, pensato e creato per scambiare documenti nel web, adattandole alla necessità di fare invocazioni di operazioni in remoto. Ad esempio, in un messaggio di request (Figura 5-2) si può utilizzare il body per passare la rappresentazione di una nuova risorsa da aggiungere al servizio e gli header per specificarne il tipo, o i parametri di query per specificare argomenti per l'invocazione aggiuntivi. La cosa più rilevante tuttavia è l'utilizzo dei "verb" HTTP per mappare le operazioni da eseguire sulle risorse. Sono questi a differenziare una richiesta per ottenere una risorsa da una per cancellarla; entrambe verranno inviate alla stessa URL che identifica la risorsa in questione, sarà il tipo di request a determinare l'operazione risultante.

HTTP verb	Operazione eseguita sulla risorsa	Effetto
GET	Ottiene una rappresentazione della risorsa all'URL specificato.	Safe
PUT	Sostituisce la risorsa sul server con quella passata all'interno della richiesta.	Idempotente
POST	Aggiunge l'elemento passato nella richiesta alla risorsa.	/
DELETE	Rimuove la risorsa associata all'URL specificato.	Idempotente
OPTIONS	Restituisce la lista delle operazioni invocabili sulla risorsa all'URL specificato.	Safe

Tabella 5-1- Effetti delle diverse request HTTP in REST

GET e OPTIONS sono operazioni safe, ovvero non hanno alcun effetto sulla rappresentazione della risorsa lato server; PUT e DELETE sono invece operazioni idempotenti, possono cioè essere ripetute n volte senza che il risultato cambi dopo la prima invocazione. POST potrebbe apparentemente sembrare equivalente alla PUT, ma così non è; non si tratta infatti di un'operazione idempotente, e inviare due volte di fila la stessa richiesta POST, cambia il risultato rispetto a inviarla una sola volta. Ad esempio, nel caso di una ipotetica collezione di risorse, all'URL: `http://myapp.net/resources/`, si potrebbe invocare una PUT per sostituire l'intera lista con una nuova, o si potrebbe invocare una POST, passando una nuova risorsa come argomento, per far sì che venga aggiunta alla collezione.

Ancora una volta è bene notare che queste sono linee guida fornite da REST e non c'è nessun impedimento reale alla creazione di un servizio che alla ricezione di una GET cancelli la risorsa in questione.

5.3.4 Gestione dello stato nei RESTful Web Services

Si è già discusso delle differenze tra stato di sessione e stato interno del server, e di come, quando si parla di stateless server, ci si riferisca al primo di questi due. Tuttavia ci sono numerose diverse applicazioni che possono avere bisogno di uno stato di sessione (si pensi anche solamente ad un'applicazione con alcuni servizi che possono essere acceduti solo se ci si è autenticati con un servizio di login), e ad un'analisi superficiale si potrebbe pensare che il modello REST non permetta in assoluto il mantenimento di stato di sessione. Questa assunzione tuttavia non è corretta.

Lo stato di sessione non viene mantenuto sul server, ma può essere mantenuto sui singoli client. Interagire con uno stateless server significa che, ad ogni richiesta, può rispondere un'istanza diversa del server stesso in modo del tutto trasparente: non c'è session affinity o sticky session. Il client si occupa di mantenere lo stato di sessione e passarlo all'interno della successiva chiamata al server.

È da qui infatti che viene l'*ST* della sigla *REST*: State Transfer, trasferimento di stato. Lo stato viene trasferito avanti e indietro tra client e server e potenzialmente

aggiornato ad ogni interazione, piuttosto che avere un server che si occupi di mantenere e gestire la sessione di tutti i suoi client. Questo è un modo intelligente per scalare su un numero estremamente elevato di utenti contemporanei; la gestione delle sessioni non pesa più sul server ma viene distribuita e ammortizzata su tutti i client, e il server può essere più facilmente replicato per servire molti più clienti concorrentemente.

Questo aspetto è parte importante dei motivi per cui i RESTful Web Services hanno raggiunto una diffusione e un successo molto maggiori rispetto ai poco usati Web Services Stateful.

Questo modello è possibile tuttavia solo se si soddisfa un vincolo fondamentale: lo stato deve poter essere serializzabile per poter essere trasferito.

5.3.5 Analisi comparativa

In modo analogo ai WS tradizionali i RESTful WS continuano a garantire disaccoppiamento tra client e server, un'alta disponibilità e riusabilità del codice e continuano a soffrire di inefficienza del protocollo di comunicazione, sebbene l'overhead sia ridotto per l'assenza dell'incapsulamento SOAP del messaggio, dei file di descrizione WSDL e dell'infrastruttura UDDI. Inoltre si registrano i seguenti punti di forza e punti deboli aggiuntivi.

Vantaggi:

- **Estrema semplicità:** l'implementazione di un server REST è estremamente semplice e consiste principalmente nel mapping di metodi (anche pre-esistenti) su risorse e metodi HTTP. Anche l'utilizzo da parte del client dei servizi offerti è semplice quanto effettuare una richiesta HTTP su un'URL.
- **Compatibilità:** l'utilizzo di servizi REST ha come unico prerequisito l'accesso ad un client HTTP (anche un semplice browser). Questo li rende utilizzabili su dispositivi e linguaggi estremamente eterogenei.
- **Scalabilità:** i server sono stateless e possono essere replicati per scalare su un numero molto elevato di utenti mantenendo bassi i costi.

Svantaggi:

- **Limitatezza di HTTP:** sebbene l'utilizzo di HTTP contribuisca attivamente a tutti i pro precedentemente elencati, è anche vero che HTTP non è un protocollo nato e pensato per l'invocazione di servizi e lo scambio di messaggi e pertanto comporta delle limitazioni. Ad esempio è impossibile mappare più di quattro interazioni differenti su una stessa risorsa (escludendo i metodi di controllo).
- **Overhead legato allo State Transfer:** il trasferimento di stato permette di mantenere il server stateless, tuttavia in caso di stati di sessione molto estesi, il trasferimento dello stesso ad ogni interazione tra client e server ha un costo notevole a livello di comunicazione. Il problema è amplificato dall'inefficienza di HTTP come protocollo per lo scambio di messaggi e da quella dovuta alla rappresentazione in formato testuale dei dati.

5.4 Java EE e EJB

Java EE (o Java Platform Enterprise Edition) è un'estensione di Java SE che definisce una specifica molto articolata per lo sviluppo di applicazioni Java di livello enterprise e per la realizzazione dell'ambiente di esecuzione in cui queste applicazioni devono eseguire.

Java EE si basa su un modello component-container, secondo il quale le applicazioni sono costituite da componenti che possono eseguire solo all'interno di un container; questo si occupa in modo trasparente di varie mansioni, come ad esempio (nel caso specifico di Java EE), la gestione del ciclo di vita dei componenti, la gestione della concorrenza, replicazione e fault-tolerance, transazionalità, persistenza, sicurezza, gestione e iniezione delle dipendenze.

È importante notare come, sebbene simile per scopo all'idea di framework o di middleware, un modello a container si differenzia per il fatto che molte delle funzionalità che offre vengono realizzate in modo trasparente e autonomo dal container stesso, e non devono essere chiamate esplicitamente all'interno del codice del componente.

Questo è possibile principalmente tramite due meccanismi principali: configurazione e intercezione delle chiamate.

Lo sviluppatore scrive la sola logica di business all'interno del suo componente e specifica in modo dichiarativo (in passato tramite descrittori XML, nelle ultime versioni quasi esclusivamente tramite annotazioni) le proprietà aggiuntive che desidera (ad esempio transazionalità), a questo punto quando un cliente richiederà un'operazione il container, in modo trasparente intercetterà la chiamata in entrata e la risposta in uscita occupandosi di aggiungere la logica transazionale.

In Java EE la funzione del container è incarnata dal cosiddetto Application Server di cui esistono diverse implementazioni realizzate da diversi produttori che, tuttavia, devono comunque rispettare la specifica per essere considerati Java EE compliant.

I principali componenti per la realizzazione di logica di business sono invece gli EJB (Enterprise Java Beans) che in realtà esistono in più versioni:

- **Stateless:** EJB privo di stato di sessione; è pertanto facilmente replicabile e scala estremamente anche con un numero molto elevato di clienti. Essendo stateless un cliente non può contare sul fatto che a richieste successive risponda sempre la stessa istanza del bean.
- **Stateful:** EJB simile a quello stateless ma che può mantenere stato di sessione; più complicato da replicare e meno scalabile della controparte senza stato. Quando un cliente invia richieste successive può contare sul fatto che risponda la stessa istanza del bean (in realtà può essere un'istanza diversa alla quale però viene fornito lo stato di sessione corretto in modo trasparente dal container).
- **Singleton:** EJB che può mantenere stato, ma di cui esiste una sola istanza all'interno del container. È solitamente utilizzato per realizzare servizi di sistema, mantenere stato condiviso, e realizzare operazioni di startup. L'accesso concorrente ai suoi metodi può essere regolato specificando se i metodi sono di sola lettura (accesso concorrente permesso) o anche di scrittura (accesso serializzato).

- **Message-driven:** EJB sensibilmente diverso dagli altri e legato a JMS; si occupa di ricevere e processare messaggi JMS in modo asincrono. Non può essere invocato o richiamato direttamente, ma viene attivato solo dal container alla ricezione di un messaggio.

L'accesso da parte di un client ad un bean avviene in modo indiretto, invocando operazioni su un oggetto proxy del bean ottenuto tramite un servizio JNDI su cui il server deve averlo precedentemente registrato. Questo garantisce un certo livello di disaccoppiamento, il cliente infatti non deve necessariamente conoscere direttamente il server, ma solo il nome logico del bean all'interno del server JNDI.

5.5 JSON Web Token

JWT (JSON Web Token)^[JWT] è uno standard open^[JRFC] che specifica come creare token di accesso basati su JSON e pertanto particolarmente indicati all'uso in RESTful Web Services. All'interno di un JWT può essere specificato qualsiasi numero e tipo di proprietà, un uso realistico ad esempio potrebbe essere quello di un server che in risposta ad una richiesta di autenticazione genera un JWT contenente un "claim" contenente il ruolo dell'utente e glielo restituisce. Ad ogni interazione successiva l'utente fornisce il token al server, per provare il proprio livello di accesso. JWT si appoggia e integra altri standard relativi a JSON come JWS (JSON Web Signature)^[JWS] per la firma e verifica di dati in formato JSON, e JWE (JSON Web Encryption)^[JWE] per criptare e decriptare dati in formato JSON.

5.5.1 Struttura dei token

Un token JWT si presenta come una stringa alfanumerica divisa in tre parti separate da punti, ad esempio:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJz  
dWIIoiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gR  
G9lIiwiaWF0eWVRtaW4iOiNyZWV9.TJVA95OrM7E2cBab3  
0RMHrHDcEfxjoYZgeFONfh7HgQ
```

Tabella 5-2- Rappresentazione testuale di un JWT

Ogni parte rappresenta un elemento JSON codificato in Base64URL.

La prima parte è l'encoding dell'header: un oggetto JSON che contiene una serie di campi per l'identificazione e decodifica del token stesso. Decodificando l'header dell'esempio precedente si ottiene:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9	{ "alg": "HS256", "typ": "JWT" }
--------------------------------------	---

Tabella 5-3- Versione codificata in base64URL e decodificata di un header JWT

Un oggetto JSON contenente due proprietà: “alg” che indica l'algoritmo di hashing usato per firmare il token (in questo caso HS256 sta per HMAC-SHA256), e “typ” che indica il tipo del token.

La seconda parte è il payload: un oggetto JSON contenente una serie di campi detti “claim”. Questi possono essere:

- **Riservati:** un insieme di claims predefiniti, non obbligatori, ma raccomandati per fornire una serie di informazioni utili all'interno del JWT. Alcuni di questi sono “iss” (issuer), “exp” (expiration time), “sub” (subject), “aud” (audience), “iat” (issued at).
- **Pubblici:** questi claims sono definiti a piacimento dall'utente, ad esempio per contenere le informazioni sul ruolo di un utente o dello stato di sessione. Tuttavia questi sono pubblici e pertanto devono essere definiti e registrati per evitare collisioni nei nomi dei token.
- **Privati:** questi claims sono definiti a piacimento dall'utente, ma a differenza di quelli pubblici hanno valenza locale, ristretta a una serie di server e client che si sono accordati sul loro uso. Per questi motivi non hanno restrizioni sui nomi.

Decodificando il payload dell'esempio precedente si ottiene:

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiYWRtaW4iOnRydWV9	{ "sub": "1234567890", "name": "John Doe", "admin": true }
--	--

Tabella 5-4- Versione codificata in base64URL e decodificata di un payload JWT

La terza ed ultima parte del token JWT è la signature. Questa è composta a partire da: la codifica dell'header, la codifica del payload, un segreto e l'algoritmo specificato nell'header. Questi elementi vengono composti calcolando la codifica base64 di (se l'algoritmo usato è HMAC-SHA256 ad esempio):

Codice 5-1- Metodo di composizione generale di una signature JWT

TJVA950rM7E2cBab3 0RMHrHDcEfXjoYZge FONFh7HgQ	HMACSHA256("eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9" + ". " + "eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRydWV9" ,"secret")
---	---

Tabella 5-5- Versione codificata in base64URL e decodificata di una signature JWT

Queste tre parti vengono concatenate separate da un punto e rappresentano la serializzazione del JWT. Questa è in tutto e per tutto una normale stringa e può essere a sua volta firmata ed usata come payload di un JWS o criptata in un JWE.

5.5.2 Sessione con token e differenza con i cookies

Il sistema a token implementato da JWT si presta ovviamente molto bene al mantenimento dello stato di sessione in un ambiente come quello dei RESTful Web Services, sia per la gestione dell'autenticazione sia per altri tipi di stato.

Molte volte tuttavia il sistema a token viene confuso con quello a cookies, con il quale effettivamente possono esserci alcune somiglianze. I due sistemi tuttavia sono intrinsecamente diversi.

Il modello a token è piuttosto semplice e permette di avere sessioni anche su server completamente stateless, mantenendo lo stato lato client.

Una semplice interazione di autenticazione tra client e server può avvenire come segue:

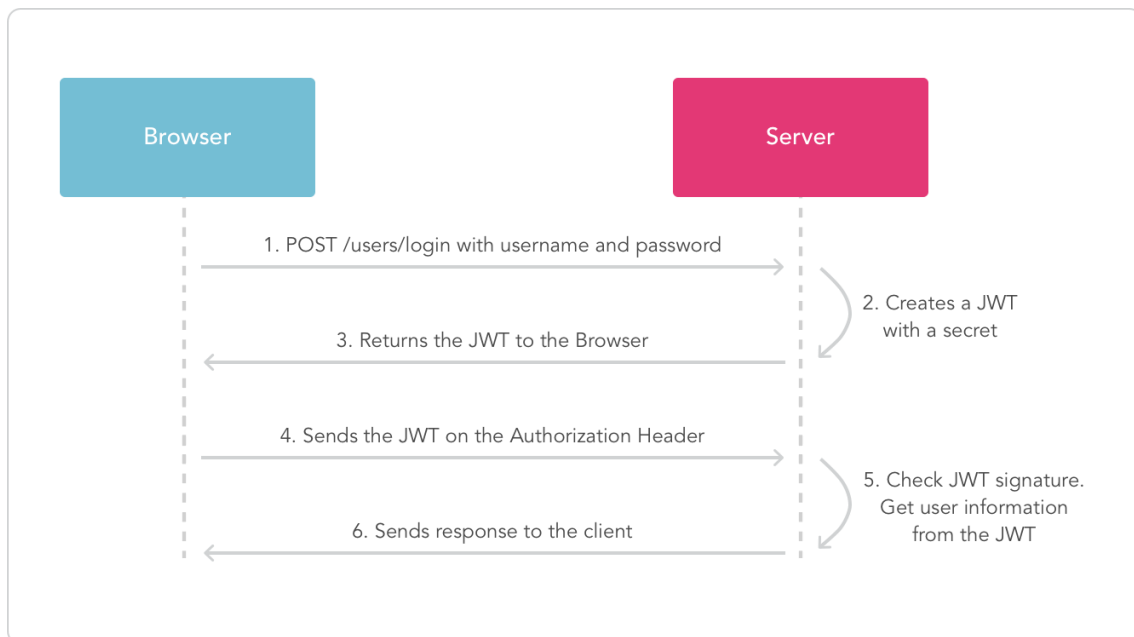


Figura 5-3- Rappresentazione di una tipica interazione di autenticazione tra client e server con sistema di autenticazione a token

1. L'utente invia le proprie credenziali al server
2. Il server verifica le credenziali dell'utente e quindi crea e firma un token.
3. Il server invia il token in risposta all'utente. Questo token deve essere mantenuto dal cliente nel modo che preferisce, in memoria, su file, ecc...
4. Successive richieste del cliente al server devono contenere questo token nel modo richiesto, sia esso nel corpo della richiesta, nell'header o come parametro all'interno dell'URL.
5. Il server controlla la firma del token e se è valida, ottiene le informazioni al suo interno necessarie per identificare l'utente.

6. Il server risponde al client; se le informazioni all'interno del token sono cambiate, si occuperà di inviare una copia aggiornata dello stesso.
7. Quando un utente deve fare log-out, questo avviene solamente lato client; è sufficiente cancellare il token, nessuna ulteriore interazione con il server è necessaria.

Sebbene il concetto alla base dell'autenticazione tramite cookies sia simile, questa avviene, al contrario di quella tramite token, in modo stateful. Ciò significa che lo stato di sessione viene mantenuto in modo combinato sia dal cliente che dal servitore. Il server salva una sessione attiva caricandone lo stato su un database e creando un cookie che ne contiene l'identificatore necessario per recuperarla. Una possibile interazione di autenticazione avviene quindi come segue:

1. L'utente invia le proprie credenziali al server
2. Il server verifica le credenziali dell'utente e quindi crea una sessione che viene salvata su database
3. Un cookie contenente l'ID della sessione viene salvato lato client
4. In successive richieste l'ID contenuto nel cookie viene utilizzato per recuperare la sessione su database e, se questa è valida, si procede a processare la richiesta
5. Quando il cliente desidera fare log-out, la sessione deve essere distrutta sia su client che su server.

Quindi sebbene si possa dire che entrambi i metodi utilizzano un sistema che utilizza token da passare al cliente e che il cliente deve presentare al server per identificarsi, un cookie contiene al suo interno solamente un identificatore, ma lo stato e i dati della sessione sono comunque salvati e gestiti lato server, mentre in JWT, ad esempio, il server è stateless ed è il token stesso a contenerne lo stato.

6. Scelte tecnologiche

6.1 Interfaccia utente e business logic

6.1.1 *Web Services*

Il modello dei Web Services, al di là delle specifiche interpretazioni, si presta bene alla realizzazione delle interfacce di accesso al servizio. I Web Services garantiscono la raggiungibilità del servizio sia in una rete locale che potenzialmente su internet, identificandolo in modo semplice tramite una URL. La comunicazione avviene tramite scambio di messaggi, appoggiandosi su protocolli di comunicazione ampiamente usati e supportati da numerosi device in commercio.

Questa struttura a scambio di messaggi permette inoltre di disaccoppiare fortemente clienti e servitori, fornendo un'interfaccia di accesso uniforme che astrae dalla realizzazione del servitore stesso, e che consente a clienti eterogenei di accedere al servizio.

Web Services sono quindi una soluzione plausibile per l'implementazione delle interfacce di accesso, soddisfacendo diversi dei requisiti posti per la realizzazione del server.

Per scegliere tra le due principali interpretazioni di Web Service, SOAP-WSDL e REST, è necessario valutare quale dei due si adatti meglio ai requisiti della applicazione server in esame.

- **Leggerezza:** i Web Services basati su SOAP e WSDL hanno un modello di comunicazione più strutturato ma significativamente più pesante di quelli basati su REST. L'uso estensivo di XML per il formato dei messaggi e per le interfacce, e il notevole overhead nella comunicazione, rappresentano uno svantaggio notevole in un ambiente intrinsecamente leggero e dinamico come quello di IoT, e in particolare dello scenario in esame.

- **Stato di sessione:** il server deve poter supportare interazioni con stato di sessione. I WS classici possono essere creati in una variante stateful utilizzando l'estensione WS-Addressing; i servizi RESTful sono stateless, ma mantengono stato di sessione sui clienti trasferendolo ad ogni interazione. Il modello REST è quello che ha avuto più successo e i WS stateful non possono contare su una implementazione standard. Tuttavia in REST lo stato di sessione deve essere completamente serializzabile per essere trasferito.
- **Disaccoppiamento:** sia i WS classici che i quelli RESTful garantiscono disaccoppiamento tramite il modello a scambio di messaggi e l'uso di un'interfaccia che nasconde i dettagli implementativi. I servizi RESTful garantiscono tuttavia copertura di un maggior numero di client, grazie alla possibilità di essere utilizzati da qualsiasi dispositivo abbia un client HTTP.
- **Scalabilità (opzionale):** in particolar modo in caso di interazione stateful, un'implementazione RESTful del servizio scala sensibilmente meglio di una SOAP-WSDL. Il server RESTful rimane infatti stateless, garantendo una migliore scalabilità.

Considerando tutti questi punti, un'interfaccia realizzata tramite Web Services REST sembra la scelta migliore.

I WS tuttavia offrono delle linee guida esclusivamente per il livello di presentazione, meccanismi di accesso a servizi per clienti e meccanismi di pubblicazione dei propri servizi per il server; il modello non contiene nulla riguardo la logica di business, l'implementazione vera e propria dei servizi. Questa è dunque lasciata interamente allo sviluppatore che può realizzarla a suo piacimento, utilizzando le tecniche, i linguaggi, i framework e l'architettura server che preferisce. Sebbene questo garantisca grande libertà e verosimilmente grandi ottimizzazioni, la realtà è che se i servizi offerti superano un certo livello di complessità ci si scontrerà con problematiche classiche quali la gestione della

persistenza e degli accessi a database, gestione della concorrenza, gestione della sicurezza, scalabilità, e altre.

Queste problematiche sono ovviamente risolvibili con codice e soluzioni ad-hoc: tuttavia ciò, oltre ad essere estremamente time-consuming, apre ulteriori problematiche legate, ad esempio, all'estensibilità e alla riusabilità del codice, in caso si voglia cambiare qualcosa della propria applicazione server in futuro.

Di conseguenza spesso si ricorre a middleware software di supporto, che sollevino lo sviluppatore dalla responsabilità di occuparsi di queste problematiche comuni, lasciandogli il solo compito di occuparsi della logica di business di livello più alto.

Sebbene la soluzione di usare Web Services REST per l'interfaccia utente, combinata con una logica di business che si appoggia su middleware sembrerebbe adatta al caso in esame, bisogna considerare che d'altra parte l'uso di middleware solitamente comporta un appesantimento dell'intera applicazione e una maggiore richiesta di risorse. Ciò potrebbe danneggiare le performance della nostra applicazione che deve poter eseguire su dispositivi poco potenti.

Questa soluzione quindi, sebbene apparentemente la migliore, potrebbe essere praticabile solamente se il middleware fosse abbastanza leggero da non impattare in modo eccessivo sulle prestazioni.

6.1.2 Java EE e EJB

Java EE fornisce mezzi adeguati per lo sviluppo di un'applicazione lato server, sia di piccole dimensioni che per grandi applicazioni distribuite su più nodi. Nel caso specifico di questa applicazione l'utilizzo di Java EE garantirebbe interazioni sia stateless che stateful, un sistema di gestione della persistenza e della sicurezza estremamente semplificato (grazie al supporto offerto dal modello a container), discreto disaccoppiamento (per altri dettagli si veda il paragrafo successivo), e un'ottima scalabilità.

Due punti che possono risultare controversi tuttavia sono i seguenti:

- **Leggerezza:** Java EE ha la fama di essere piuttosto pesante come sistema e quindi si potrebbe pensare che non sia adatto alla realizzazione

un'applicazione destinata (anche) a dispositivi con prestazioni limitate. Questa fama in realtà non è meritata, ed è dovuta principalmente alle vecchie versioni 1.4 di Java EE e 2.0 di EJB (attualmente Java EE è arrivato alla 7 ed EJB alla 3.2). Anche in questo caso inoltre la pesantezza era riferita in particolare al modello di sviluppo delle applicazioni (che faceva largo uso di descrittori di deployment scritti in XML), piuttosto che ad una pesantezza prestazionale. La realtà è che la differenza di memoria utilizzata da una applicazione realizzata con JEE rispetto ad una realizzata con POJO è così piccola da essere trascurabile, e può essere ulteriormente ridotta con un'adeguata configurazione dell'Application Server (e anche lo sviluppo è diventato semplice e veloce da EJB 3.0 in poi). Inoltre, negli ultimi tempi si è notata una tendenza a realizzare versioni micro (Payara Micro) o modulari (Wildfly Swarm) degli Application Server più popolari, ulteriormente ottimizzati per l'esecuzione su dispositivi limitati.

- **Interfaccia di accesso e disaccoppiamento:** il metodo di accesso ad EJB garantisce un certo grado di disaccoppiamento tramite l'uso di JNDI e di oggetti proxy, tuttavia la necessità di un server JNDI rende inadatto questo tipo di accesso al nostro scenario applicativo. Inoltre l'implementazione del server non è totalmente trasparente ed è richiesto che il cliente utilizzi Java, limitando notevolmente la possibilità di avere clienti altamente eterogenei. EJB fornisce sin dalle sue prime versioni la possibilità di esporre alcuni dei propri bean come Web Services standard; sebbene questo rappresenti un netto miglioramento ci si scontra ancora una volta con i problemi di WS standard, elencati precedentemente.

Dunque sebbene Java EE rappresenti un'ottima alternativa per la realizzazione della logica di business dell'applicazione, il modello di accesso non è funzionale ai nostri scopi, almeno nella sua forma base. La possibilità di usare un'interfaccia WS è sicuramente interessante, tuttavia esiste un'alternativa possibilmente migliore.

6.1.3 Ibridazione REST-EJB

A partire da Java EE 6 e in particolare dalla versione 3.1 di EJB, è divenuto possibile esporre in modo diretto uno stateless bean (o un singleton bean) della propria applicazione EJB come RESTful Web Service in modo semplice e immediato. Questo ibrido EJB/RESTful WS, permette di attingere ai vantaggi di entrambe le tecnologie, in particolare:

- **Accesso semplice via http:** lo stateless bean diventa pienamente utilizzabile come servizio RESTful e pertanto i suoi metodi possono essere acceduti da qualsiasi dispositivo disponga di un client http, alla rispettiva URL.
- **Injection:** resta possibile utilizzare l'injection per iniettare risorse nell'EJB. Si possono iniettare altri EJB, Entity Manager, risorse JMS o Data Source. Queste diventano tutte liberamente utilizzabili dal servizio RESTful.
- **Transazioni:** le modifiche apportate in una chiamata REST verranno automaticamente sincronizzate con il database in modo trasparente dal container, proprio come avviene in un normale EJB.
- **Single threaded programming model:** La concorrenza e la gestione dei thread viene completamente gestita dal container. Gli EJB sono thread safe e questa proprietà viene ereditata da questo ibrido RESTful. La programmazione può avvenire senza curarsi minimamente degli aspetti di multi-threading. Si può far scalare semplicemente il servizio ampliando o restringendo a piacere i pool di bean nelle impostazioni del server.
- **Monitoring:** Un EJB, e quindi anche la sua versione ibrida, è visibile in JMX.
- **Altre funzionalità proprie di EJB:** Ad esempio: metodi interceptor. L'intercezione delle chiamate funziona correttamente anche con l'ibrido EJB/REST, si può facilmente esternalizzare logica al di fuori del codice del servizio REST, per effettuare controlli sui parametri di invocazione o sui valori di ritorno proprio come si farebbe con un EJB (ad esempio per

controllare l'autorizzazione del cliente prima dell'esecuzione di un metodo).

Anche la realizzazione è piuttosto semplice: se il punto di partenza è un EJB (stateless o singleton, non stateful!) è sufficiente decorare il codice con le annotazioni tipiche di JAX-RS^[JRS] (@Path, @GET, @POST, @Consumes, @Produces, ecc...), se il punto di partenza è invece un servizio RESTful, sarà sufficiente aggiungere l'annotazione @javax.ejb.Stateless alla classe e si potranno da subito utilizzare le funzionalità discusse sopra (chiaramente si presuppone che sia già presente tra le dipendenze la libreria javaee versione 6 o superiore).

Questo approccio risulta quindi essere il più adatto al nostro caso; ci permette di utilizzare Java EE per la parte di business logic soddisfacendo, come già mostrato, i requisiti in quell'ambito, e consente di realizzare l'interfaccia d'accesso tramite servizi REST (risolvendo i problemi di disaccoppiamento e pesantezza di EJB in questo settore), potenziati per altro dall'integrazione con le funzionalità di Java EE.

6.1.4 Implementazioni disponibili

L'Application Server scelto per ospitare l'applicazione è Payara^[PAY], fork open source sviluppato e mantenuto dalla community del più famoso Oracle Glassfish 4^[GLS]. Payara è Java EE 7 compliant, corregge molti problemi che notoriamente affliggono Glassfish, riceve aggiornamenti e bug fix regolari, è open source e offre una versione secondaria chiamata Payara Micro^[PYM] pensata appositamente per eseguire su dispositivi poco potenti (ad esempio Raspberry Pi). Payara, e in particolar modo Payara Micro, utilizzano una quantità estremamente limitata di risorse^[PMEM] risultando come un'ottima scelta per lo scenario applicativo in esame. Per la realizzazione dei RESTful Web Service viene utilizzata l'implementazione standard di Payara della specifica JAX-RS^[JRS] che, essendo compresa in Java EE, deve essere realizzata dall'Application Server. Payara utilizza la reference implementation di JAX-RS: Jersey^[JSY].

6.2 Rappresentazione dei dati

I Web Service RESTful si appoggiano su HTTP e questo significa che i dati scambiati tra client e server devono in qualche modo essere contenuti all'interno delle request e response HTTP. Essendo questi messaggi di tipo testuale, i dati che si intende trasferire dovranno necessariamente essere serializzati o rappresentati in formato testuale.

6.2.1 *JSON e XML*

I due principali formati utilizzati come standard de-facto in REST sono JSON e XML, entrambi testuali, entrambi in grado di rappresentare oggetti ed entrambi soddisfano gran parte delle necessità per il trasferimento dati con differenze reali minime. In generale si tende a preferire JSON per il trasferimento dei dati, poiché a parità di informazione rappresentata è meno verboso della sua controparte, mentre si preferisce XML per la rappresentazione di strutture complesse (come la descrizione di un servizio), perché la sua natura di linguaggio a markup lo rende più strutturato e navigabile. Considerato ciò, non c'è motivo di scegliere in assoluto uno o l'altro, e si può piuttosto utilizzarli entrambi a seconda della specifica necessità.

6.2.2 *Librerie per la manipolazione dei dati*

Per XML le librerie standard Java offrono due soluzioni: JAXP (Java API for XML Processing)^[JXP] e JAXB (Java Architecture for XML Binding)^[JXB]. La prima è una libreria che permette di costruire documenti XML, validarli, e farne il parsing, JAXB invece consente di gestire il mapping di classi Java su documenti XML in modo automatico (dopo una necessaria configurazione, solitamente tramite annotazioni). Esistono poi numerose librerie esterne realizzate per la gestione dei file XML; in particolare per l'applicazione si è scelto JDOM, una libreria che racchiude in sé funzionalità simili a quelle offerte da JAXP e JAXB, ma che fornisce una sintassi più leggera, più adatta per le necessità limitate dell'applicazione.

JSON non può ancora contare, invece, su supporto nativo all'interno del JDK, tuttavia sono state definite delle API standard per la manipolazione di JSON in Java tramite il JSR 353^[J353], e un'implementazione standard delle stesse: JSON Processing^[JSONP]. In realtà, considerata la diffusione sempre maggiore del formato, esistono numerose altre alternative, anche open source, per la gestione di JSON in Java; tra le più popolari troviamo Google GSON^[GSON] e Jackson^[JKSN]. Considerando le prestazioni^[JPER] delle varie soluzioni, risulta che JSON-P rimane piuttosto indietro in termini di performance sia rispetto a GSON che Jackson, mentre quest'ultimo ottiene risultati migliori per file di grandi dimensioni e GSON risulta più adatto per scambi numerosi di piccoli file. Tutto considerato la soluzione che sembra più appropriata al nostro scenario applicativo è dunque GSON.

Guardando a possibili sviluppi futuri per il supporto JSON in Java, sembrava inizialmente che dovesse essere rilasciata una libreria per il supporto nativo a JSON nel prossimo JDK 9^[JEP], ma questa è stata successivamente rimossa e rimandata ad un eventuale JDK 10 o 11. Lato Java EE invece, è prevista l'implementazione di JSON-B (Java API for JSON Binding)^[JBIN], nella prossima versione Java EE 8, una serie di API similari a JAXB, che consentono di gestire il binding di classi Java in formato JSON.

6.3 Sicurezza

Utilizzando interfacce utente di tipo REST, diventa necessario utilizzare meccanismi adeguati e pensati specificatamente per questo tipo di servizi anche per la sicurezza. L'esempio più banale di questa necessità è il login con determinate credenziali e l'esecuzione delle successive richieste senza dover reinviare le credenziali complete ad ogni invocazione, ma mantenendo una sessione di autenticazione.

Come già discusso il mantenimento di una sessione con un servizio RESTful non è banale, essendo che il server non mantiene alcuno stato. Lo stato di sessione

quindi deve essere mantenuto dai singoli clienti, ovviamente con le dovute misure di sicurezza per garantirne integrità e autenticità.

6.3.1 JSON Web Token o cookies

In generale JWT può contare diversi vantaggi su quello a cookie:

- **Stateless, scalabili e disaccoppiati:** come detto nel capitolo precedente, il vantaggio più significativo dei token sui cookies è che il server può essere completamente stateless, permettendo di scalare molto meglio rispetto ad uno stateful. L'unico compito del server diventa quello di firmare i token e verificarli quando gli vengono inviati dal client.
- **Possibilità di trasferire stato nel JWT:** un cookie può contenere al massimo 4KB di dati, mentre un JWT può contenere stato complesso fintanto che è rappresentabile in JSON, mantenendo l'utilità di conservarlo lato client, cosa particolarmente utile in ambito REST.
- **Performance:** usando un sistema di autenticazione a cookie le sessioni attive devono essere salvate su database, e questo si traduce in un accesso a database ad ogni richiesta con stato di sessione. Questo può essere particolarmente impattante sulle performance dell'applicazione, specie nel nostro scenario applicativo. Un JWT d'altra parte contiene tutte le informazioni necessarie, limitando le chiamate necessarie lato server.
- **Non browser-centrici:** i cookie sono nati e pensati per l'interazione con i browser, e non si integrano bene con un sistema come REST, in cui HTTP viene utilizzato per effettuare invocazioni remote dai dispositivi più disparati, piuttosto che solamente da browser. JWT d'altro canto sono pensati specificatamente con questo scenario in mente, rendendoli molto più adatti all'uso in applicazioni IoT.

L'unico possibile vantaggio, in questo caso, dei cookies è il minore overhead sui canali di comunicazione. Un session cookie è, in generale, più leggero anche del più piccolo JWT che si possa pensare, e dovendoli inserire su ogni richiesta, in alcuni casi l'overhead potrebbe diventare un problema tangibile.

Nonostante quest'ultimo punto, per tutti i motivi precedentemente elencati, JWT risulta essere una soluzione più adatta.

6.3.2 Implementazione JWT utilizzata

JWT può contare su un gran numero di implementazioni^[JMPL] per i linguaggi più disparati. Per la nostra applicazione è necessaria una implementazione in Java che, per completezza, supporti tutti i claim riservati e algoritmi della specifica e che, al suo interno, fornisca anche supporto a JWS e JWE. Una implementazione che risponde a tutti questi requisiti è jose4j^[j4j], che dunque risulta essere quella scelta per la realizzazione dell'applicazione.

6.4 Persistenza

Come già detto, le esigenze del server in termini di persistenza sono piuttosto limitate. Proprio per questo motivo risulterebbe un inutile spreco di risorse orientarsi verso una soluzione standalone quando Payara integra già al suo interno un'implementazione di Apache Derby^[DRB] pronta all'uso. Apache Derby è un database relazionale implementato interamente in Java, pensato per essere estremamente light-weight, e questo si sposa perfettamente con le necessità della nostra applicazione. Se si utilizza Java EE per la parte di business logic, si possono utilizzare le funzionalità del container e in particolare JPA^[JPA], per l'accesso semplificato al database.

Tramite JPA non è necessario scrivere codice ad-hoc per il mapping object-relational degli oggetti che si vogliono salvare su database, ma è sufficiente annotare correttamente le classi che si vogliono persistere (chiamate entity), operazione facilmente realizzabile anche su pre-esistenti POJO non pensati con questo scopo in mente.

Oltre a questo JPA si occupa di gestire il ciclo di vita e la persistenza delle entity, la connessione al database, e la transazionalità delle operazioni in modo interamente trasparente, rendendo possibile operare sulle entity come se fossero semplici POJO.

Payara utilizza EclipseLink^[ELK] come provider JPA, ma può essere configurato anche per utilizzarne altri (ad esempio Hibernate). Tuttavia, ancora una volta, le necessità in ambito di persistenza di questa applicazione sono limitati e utilizzare il provider JPA standard è più che sufficiente.

6.5 Motore Prolog

6.5.1 *tuProlog*

tuProlog^[2PL] è un framework Prolog leggero e open source, originariamente realizzato in Java, ma che attualmente supporta versioni per numerose altre piattaforme (Android e Microsoft .NET tra le altre). Lo scopo di tuProlog è di fornire le capacità di un motore Prolog in modo modulare, dinamico e configurabile; il core del sistema è infatti estremamente leggero e minimale, tuttavia può essere arricchito di funzionalità aggiuntive tramite il caricamento, in modo sia dinamico che statico, di librerie: collezioni, definibili anche dall'utente, di predicati, operatori e funtori.

tuProlog supporta nativamente la programmazione multi-paradigma realizzando un'integrazione tra Prolog e i linguaggi object-oriented per cui è stato sviluppato (Java, C# e gli altri linguaggi .NET). Questa integrazione avviene in entrambi i versi, consentendo ad esempio in Java l'uso di classi ed oggetti che permettono di configurare, utilizzare, e ottenere risultati da un motore Prolog, ma anche ad esempio di utilizzare in Prolog classi e oggetti scritti in Java direttamente al suo interno. Questo doppio aspetto consente a tuProlog di realizzare una piena interoperabilità e commistione di programmazione object-oriented e logica.

tuProlog (attualmente versione 3.2.0) è un progetto sviluppato presso il DISI dell'Università di Bologna^[DISI] ed è liberamente scaricabile da Bitbucket^[2PB] o utilizzabile nei propri progetti utilizzando Maven^[2PM]. Al download è possibile scegliere tra una versione completa ed eseguibile con interfaccia grafica, e una versione più minimale, più adatta ad essere integrata in progetti terzi.

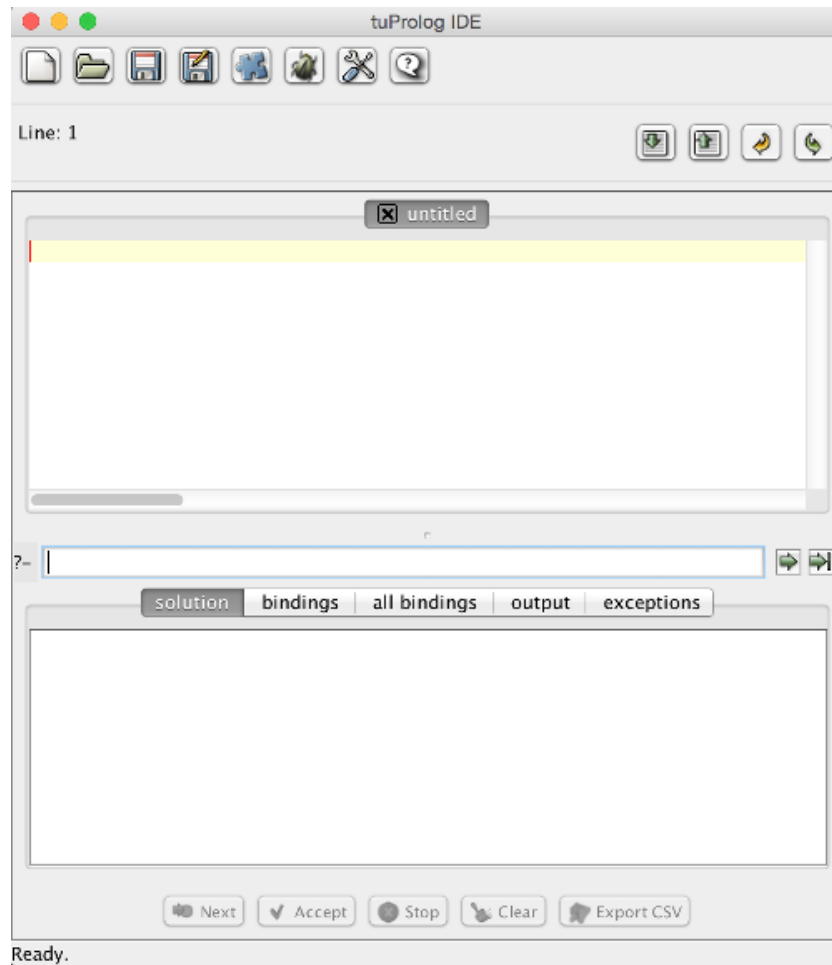


Figura 6-1- Interfaccia grafica di tuProlog

tuProlog presenta diversi vantaggi in relazione ai requisiti che abbiamo posto per la realizzazione della nostra applicazione server.

- **Piena integrazione con Java:** il motore tuProlog non solo è scritto in Java, permettendogli di essere facilmente inserito in un'applicazione Java EE ma come già detto, permette una vera e propria integrazione tra linguaggi object-oriented e Prolog.
- **Supporto nativo a JSON:** dalla versione 3.2.0 tuProlog ha introdotto metodi su molti dei suoi componenti principali (compreso l'engine stesso) per la serializzazione e deserializzazione JSON. Lo stato del motore inferenziale in questo modo risulta serializzabile e traferibile in formato testuale. Questa caratteristica risulta fondamentale per la realizzazione di interazioni con stato di sessione in un sistema basato su REST: in questo

modo lo stato dell'engine può essere mantenuto sul cliente, e ripristinato quando viene ripassato sul server.

- **Light-weight e modulare:** il core di tuProlog è estremamente leggero, cosa che si presta molto bene ad uno scenario in cui deve eseguire su dispositivi con risorse limitate. Allo stesso tempo tuttavia il core può essere integrato in modo dinamico da librerie, rendendolo potenzialmente scalabile ed efficace anche su macchine con capacità più elevate.

Per questi motivi tuProlog soddisfa i requisiti della nostra applicazione, e risulta un'ottima scelta per la realizzazione del motore inferenziale interno della stessa.

7. Progetto

7.1 Applicazione lato server

Come visto nel capitolo precedente l'implementazione del server richiede l'utilizzo di diverse tecnologie che vengono rapidamente riassunte di seguito:

- **Java EE ed EJB** → Realizzazione dei componenti della business logic
- **RESTful WS ed EJB** → Realizzazione delle interfacce
 - **Jersey (JAX-RS)** → Funzionalità REST
 - **EJB** → Integrazione con la business logic
- **tuProlog** → Realizzazione del motore inferenziale
- **Apache Derby (integrato)** → Database
 - **EclipseLink (JPA)** → Interazione con database
- **jose4j (JWT)** → Sicurezza in ambiente RESTful WS

Infine, l'intera applicazione server utilizza Maven^[MVN] per la gestione delle dipendenze e per la procedura di build.

Ricordando la volontà di creare una applicazione con livelli di astrazione differenti ciascuno diviso in moduli che realizzano funzionalità differenti (Figura 4-2), diamo un'iniziale panoramica dei componenti dell'applicazione in base alla funzionalità richiesta che realizzano, e che poi verranno discussi nel dettaglio nei successivi sotto-capitoli.

Componenti	Modulo	Funzionalità
tuPrologService	Main Interface	Web Service REST per l'interrogazione del motore d'inferenza.
configurationService	Configuration Interface	Web Service REST ad accesso limitato per la configurazione dinamica del server.
SecurityManager	Authentication and Security	Fornisce agli altri componenti funzionalità legate alla sicurezza.
AuthCheckInterceptor	Authentication and Security	Interceptor per la verifica trasparente delle autorizzazioni d'accesso degli utenti ai servizi.

StorageService	Persistence Manager	Fornisce agli altri componenti metodi controllati per l'accesso in lettura/scrittura al database.
StatelessEngine	Prolog Engine	Wrapper di un engine tuProlog che fornisce agli altri componenti metodi per interrogarlo.
EngineState	Engine Manager and Data Supplier	Contiene lo stato condiviso delle multiple istanze di statelessEngine.
DataSupplier	Engine Manager and Data Supplier	Fornisce i dati di contesto su cui lavora l'engine Prolog.

Tabella 7-1- Moduli e funzionalità dei componenti principali che realizzano l'applicazione

In particolare i componenti `tuPrologService` e `configurationService` verranno trattati nel capitolo 7.2, `SecurityManager` e `AuthCheckInterceptor` nel capitolo 7.3, `StorageService` nel 7.4, `StatelessEngine` e `EngineState` nel 7.5, e infine `DataSupplier` nel 7.6.

7.2 Interfacce di accesso al servizio

Come motivato nel capitolo precedente, si è scelto di realizzare le interfacce dell'applicazione server come RESTful Web Services ibridati con EJB, usufruendo pienamente delle funzionalità offerte da entrambi.

L'accesso via http permette di rendere il servizio disponibile praticamente a qualsiasi dispositivo contenga un browser o un generico client http, le chiamate ai singoli servizi avvengono tramite normali metodi http (GET, POST, DELETE...) e pertanto non necessitano di lookup JNDI o oggetti proxy normalmente utilizzati nei client EJB. Anche i tipi di ritorno e degli argomenti utilizzati dai servizi sono standardizzati in un'ottica web-based: viene utilizzato semplice testo dove possibile e standard come JSON e XML per la rappresentazione di dati più complessi.

Viene inoltre fatto largo uso della resource injection, per ottenere i riferimenti di altri EJB dell'applicazione che si occupano della gestione della persistenza, della sicurezza, o dello stesso motore tuProlog interno. Il salvataggio o il caricamento

dei dati di configurazione dalla memoria persistente avviene in modo trasparente, sicuro e transazionale grazie al container EJB. Si può far scalare il servizio modificando le impostazioni del server per aumentare o diminuire il pool di beans, il cui ciclo di vita viene comunque pienamente gestito dal container.

EJB torna estremamente utile anche per i controlli di autenticazione, per l'accesso ai metodi che lo richiedono; nell'interfaccia di configurazione, infatti, l'autenticazione può essere gestita in modo del tutto trasparente da un metodo interceptor nella classe `AuthCheckInterceptor`, lasciando pulito il codice dei metodi del servizio RESTful.

Gran parte dei metodi dell'interfaccia di configurazione sono semplici metodi di accesso e pertanto possono limitarsi a passare l'operazione al gestore della persistenza, che si occuperà di recuperare la configurazione richiesta o di settarne il valore sull'entity corrispondente.

Il metodo di `login` legge le credenziali dell'utente, dall'header della richiesta http e quindi utilizza il bean `SecurityManager` per ottenere il ruolo associato con quelle credenziali. Il metodo quindi fa uso nuovamente di `SecurityManager` per generare un JWT che contenga l'informazione sul ruolo assegnato e quindi per firmare e criptare il token, che infine viene restituito all'utente (per maggiori informazioni si veda il capitolo sulla sicurezza: 7.3). Anche qui le operazioni vengono esternalizzate su un altro bean, in modo da mantenere l'interfaccia ad un livello di astrazione più alto, ed evitare di inserire logiche specifiche di autenticazione, e verifica credenziali al suo interno.

Il metodo `getInfo`, comune anche all'interfaccia utente principale, utilizza la reflection e JDOM per generare in modo dinamico una rappresentazione XML dei servizi forniti dall'interfaccia, specificando nome dei metodi, URL, tipo dei parametri di ingresso e uscita, richiesta di autorizzazione e metodo http richiesto per la chiamata.

Anche parte dei metodi forniti dall'interfaccia principale (`getTheory`, `getGoalsList`, `getGoalAtIndex`) rigirano la richiesta al gestore della persistenza, mentre i metodi di tipo `getSolution` richiedono la soluzione del goal ricevuto ad

un EJB iniettato di tipo `StatelessEngine`, che gestisce l'accesso al vero e proprio motore inferenziale

7.2.1 Gestione delle chiamate con sessione

I metodi per la risoluzione con sessione devono occuparsi di gestire lo stato di sessione e quindi presentano maggiore complessità rispetto agli equivalenti senza sessione. In un modello RESTful, come già detto, non è il server a dover mantenere lo stato di sessione, ma la responsabilità ricade sui singoli client. Di conseguenza, alla fine della risoluzione del goal, il server dovrà inviare al client assieme alla risposta anche lo stato di sessione e, in questo caso particolare, lo stato serializzato dell'engine `tuProlog`.

A tal fine, nelle chiamate con sessione la comunicazione con il client avviene tramite oggetti JSON che possono contenere diverse proprietà (tra cui lo stato di sessione) e, a seconda del loro contenuto, influenzano il comportamento del metodo.

L'oggetto JSON inviato dal client deve contenere una di due proprietà:

Nome Proprietà	Contenuto Proprietà
goal	Contiene il goal di cui calcolare la soluzione. Nelle chiamate successive alla prima questa proprietà non deve essere settata, o deve essere vuota.
engine	Contiene lo stato serializzato dell'engine dopo l'ultima chiamata, firmato e criptato dal server. Il client deve settare questa proprietà con la stringa ricevuta come stato dal server durante l'ultima chiamata, senza alcuna modifica. Questa proprietà deve essere lasciata vuota o non settata alla prima chiamata della sessione, mentre va sempre settata nelle chiamate successive.

Tabella 7-2- Descrizione delle proprietà dell'oggetto JSON accettato come parametro nei metodi di tipo "getSolution" con supporto allo stato di sessione

In breve il comportamento del server a seconda delle proprietà dell'oggetto ricevuto è il seguente:

engine	goal	Risultato
Non Settato	Non Settato	412 - Precondition Failed
Non Settato	Settato	Prima chiamata della sessione.
Settato	Non Settato	Chiamata successiva alla prima.
Settato	Settato	412 - Precondition Failed

Tabella 7-3- Descrizione dei diversi risultati ottenibili, a seconda di quali proprietà sono settate o meno all'interno dell'oggetto JSON di richiesta in chiamate con sessione

Il servizio quindi ottiene dallo StatelessEngine la singola o le multiple soluzioni richieste e lo stato serializzato in JSON alla fine dell'elaborazione. Lo stato, dovendo essere inviato e mantenuto dal client, viene firmato e criptato per evitare che venga modificato o se ne possano estrarre informazioni sensibili (si veda la parte riguardante la sicurezza per maggiori informazioni a riguardo: capitolo 7.3). A questo punto la soluzione (o le soluzioni) e lo stato criptato vengono inseriti in un oggetto JSON di risposta:

Nome Proprietà	Contenuto Proprietà
solution	Contiene la soluzione o le soluzioni. Se si tratta di multiple soluzioni, queste vengono inserite come lista serializzata in JSON.
engine	Contiene una stringa rappresentante lo stato serializzato, firmato e criptato dell'engine. Il cliente dovrà conservare e inviare questa stringa come valore della proprietà "engine" dell'oggetto JSON argomento della chiamata successiva.
goal	In risposta alla prima chiamata il server restituisce anche il goal risolto come valore di questa proprietà.

Tabella 7-4- Descrizione delle proprietà dell'oggetto JSON di risposta al cliente in una interazione con sessione

7.2.2 Mapping tra metodi e risorse REST

Come detto il cliente può accedere alle interfacce come servizi RESTful, questo significa che, conformemente alle specifiche REST (in particolare ai principi ROA), i vari metodi del servizio sono mappati su risorse ad uno specifico URL, sul quale si possono effettuare operazioni diverse a seconda del metodo http invocato su quella risorsa.

Ad esempio ecco come possono essere mappati alcuni dei metodi dell'interfaccia di configurazione:

Nome Metodo	URL risorsa	Metodo http	Effetto
<code>getGoalsListAsJSON()</code>	<code>*/goalList</code>	GET	Restituisce una rappresentazione JSON della risorsa <code>goalList</code> .
<code>setGoalsList(list)</code>	<code>*/goalList</code>	PUT	La risorsa <code>goalList</code> viene impostata basandosi sulla rappresentazione JSON di una lista (<code>list</code>) passata nel corpo della richiesta PUT.
<code>addGoal(goal)</code>	<code>*/goalList</code>	POST	Aggiorna la risorsa <code>goalList</code> inserendo il goal specificato.
<code>getGoalAtIndex(i)</code>	<code>*/goalList/{i}</code>	GET	Restituisce la risorsa <code>goal</code> all'indice <code>i</code> della risorsa <code>goalList</code> .
<code>removeGoalAtIndex(i)</code>	<code>*/goalList/{i}</code>	DELETE	Elimina la risorsa <code>goal</code> all'indice <code>i</code> della risorsa <code>goalList</code> .
<code>getGoalByName(name)</code>	<code>*/goalList/byName?goal=name</code>	GET	Reindirizza all'elemento della lista contenente il goal "name".
<code>removeGoalByName(name)</code>	<code>*/goalList/byName?goal=name</code>	DELETE	Elimina la risorsa <code>goal</code> corrispondente al nome "name".

Tabella 7-5- Mapping di alcuni metodi dell'interfaccia di configurazione secondo i principi RESTful

Da notare che in caso in cui il goal venga specificato nella URL, ad esempio come parametro di query, come accade negli ultimi due casi della tabella, il client dovrà occuparsi di effettuare l'encode (in UTF-8) del valore del parametro, per permettere la corretta rappresentazione e trasferimento dei caratteri speciali.

7.3 Sicurezza

Come anticipato nel capitolo 7.1, i principali componenti che si occupano della sicurezza nell'applicazione sono:

Componente	Funzionalità
SecurityManager	EJB Singleton che si occupa di mantenere le chiavi e fornire metodi per la gestione della sicurezza agli altri componenti dell'applicazione: validazione delle credenziali, firma/validazione di dati, crittazione/decrittazione di dati.
AuthCheckInterceptor	Classe contenente un metodo interceptor che si occupa di verificare le credenziali prima dell'esecuzione di metodi ad accesso ristretto.
@RoleRequired	Annotazione che permette di specificare il ruolo richiesto per l'esecuzione del metodo che annota.

Tabella 7-6- Componenti del modulo di sicurezza dell'applicazione

Inoltre come già discusso, per la gestione della sicurezza dell'applicazione si è fatto uso di un'implementazione di JWT, una tecnologia specificatamente mirata specificatamente ai servizi REST che infatti utilizza tecnologie non estranee ai Web Service di questo tipo, come appunto JSON.

7.3.1 Autenticazione

L'autenticazione tramite JWT all'interno del servizio è realizzata dal servizio `login()`, dell'interfaccia di configurazione, mentre la verifica dei permessi ad ogni chiamata successiva è realizzata da un metodo interceptor di `AuthCheckInterceptor`, che la effettua in modo trasparente.

Il client accede al servizio di login specificando username e password nell'header del messaggio http di richiesta (N.B. Per questa versione iniziale, non si fa uso di https, e pertanto queste credenziali vengono trasmesse in chiaro. Espandendo il progetto per utilizzare https, i dati nell'header verrebbero trasmessi in modo

sicuro). Il servizio di login riceve le credenziali e ottiene il ruolo dell'utente interrogando il bean `SecurityManager` (che non fa altro che controllare sul database il ruolo dell'utente corrispondente alle credenziali). Se le credenziali non corrispondono a nessun utente, viene restituito un messaggio d'errore, altrimenti il ruolo ottenuto viene inserito come claim di un token JWT. Questo oggetto viene quindi firmato, serializzato, criptato e serializzato nuovamente utilizzando le funzionalità di JWT tramite `SecurityManager`. Il risultato è una stringa contenente la serializzazione criptata e firmata dei claims. Questo token viene restituito al cliente e sarà compito suo mantenerlo e usarlo come prova della sua avvenuta autenticazione alle chiamate successive.

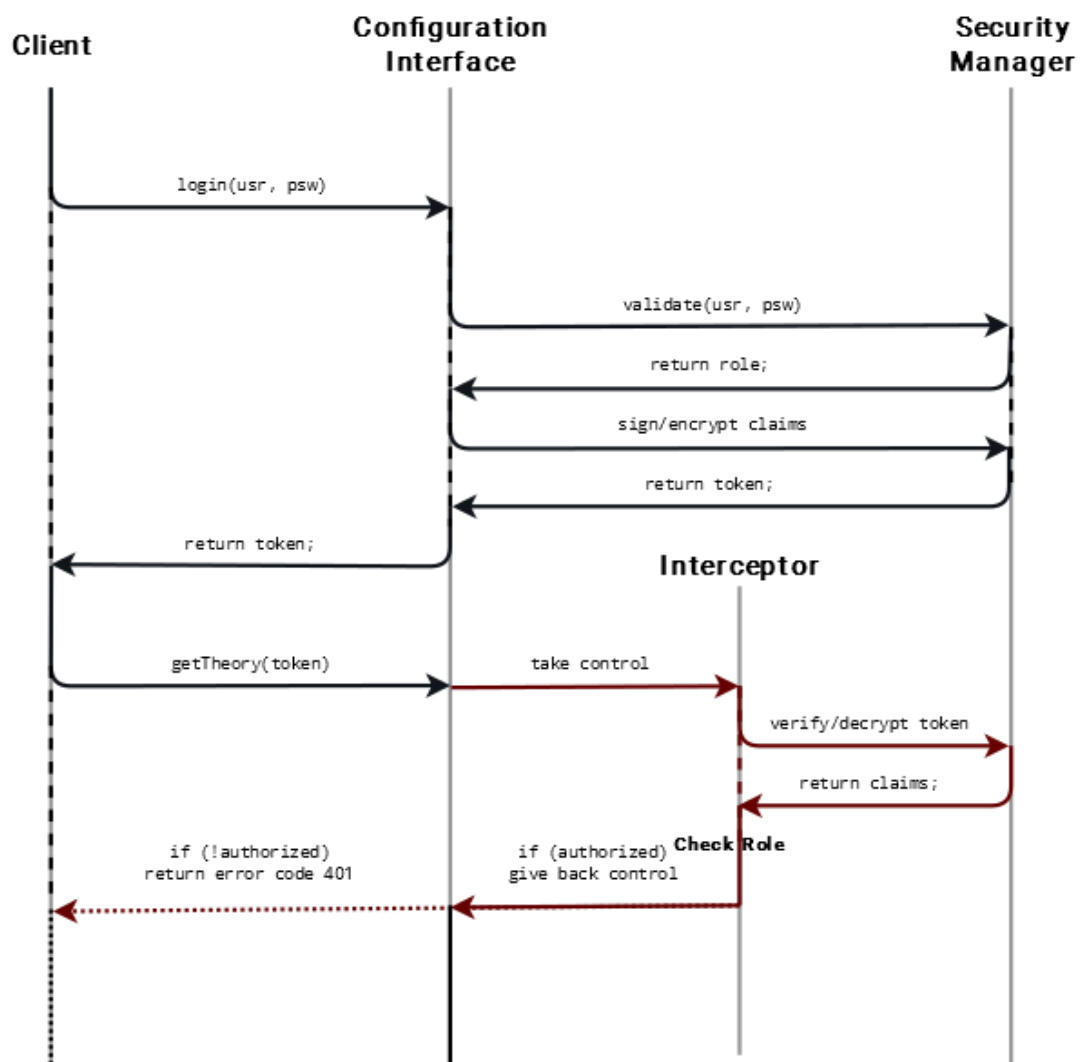


Figura 7-1- Interazione d'esempio tra client e server, in cui il client effettua il login e quindi richiede un'operazione ad accesso limitato. Vengono mostrate le chiamate scatenate tra i componenti del server alla ricezione delle due differenti richieste che realizzano le funzionalità di verifica delle credenziali durante il login e di verifica del ruolo d'accesso nelle chiamate successive.

Quando il client vorrà accedere ad una funzionalità con accesso ristretto, inserirà il token ricevuto come valore di un header con nome “token”, all’interno del messaggio http di richiesta.

A questo punto il metodo interceptor intercetterà la chiamata (Figura 7-1), verificherà se il metodo chiamato richiede autorizzazione (se è marcato con l’annotazione `@RequiresAuth(roleRequired = role)`) e in caso positivo, sempre tramite `SecurityManager`, decrypterà il token, verificherà la firma, e infine estrarrà la proprietà “role” dai claims per controllare che il ruolo dell’utente abbia autorizzazione maggiore o uguale a quella richiesta (`roleRequired`).

Se il check ha successo, l’interceptor restituisce il controllo al metodo che esegue normalmente, altrimenti invia direttamente una risposta di errore al client, bypassando il metodo.

7.3.2 Integrità e riservatezza dello stato di sessione

Altro caso in cui è necessario fornire sicurezza è nel caso, già discusso, del passaggio da server a client dello stato dell’engine Prolog, in caso di richieste di soluzione con sessione. Lo stato del motore Prolog è serializzato in JSON e pertanto, se fosse trasmesso senza ulteriori operazioni, facilmente leggibile, modificabile, e persino sostituibile dal client o da terzi che intercettino il messaggio. Ancora una volta quindi vengono utilizzate le funzionalità di `jose4j`, e la serializzazione dell’engine, prima di essere inviata al client, viene firmata e criptata, rendendone impossibile la lettura o la modifica.

Inoltre è possibile, cambiando le chiavi utilizzate per la crittazione dell’engine, invalidare in modo semplice le sessioni attualmente in corso. All’invio dell’ennesima richiesta della sessione, lo stato risulterà non più valido e la sessione avrà termine.

7.3.3 Componenti del modulo di sicurezza

La maggior parte di queste funzionalità è realizzata, come detto, dall’EJB `SecurityManager`. Si tratta di un bean che, come suggerisce il nome, si occupa delle operazioni concernenti la sicurezza; al suo interno sono mantenute le chiavi per

firmare/verificare e criptare/decriptare credenziali e stato dell'engine Prolog, oltre a metodi per effettuare queste operazioni ottenendo direttamente la serializzazione in risposta. Inoltre è sempre questo componente ad occuparsi della verifica delle credenziali appoggiandosi a StorageService per l'accesso al database.

Ad ogni utente viene assegnato un ruolo (guest, configurator, admin). In questa versione prototipale dell'applicazione risulta secondario realizzare una ipotetica "interfaccia admin" per permettere agli utenti con quel ruolo di configurare, promuovere, eliminare, aggiungere e gestire utenti. Il server può limitarsi per ora a leggere gli utenti all'avvio da database o, se il database è vuoto, da un file di configurazione XML modificabile a deploy time. Ciò nonostante è opportuno implementare il sistema interno in modo che la futura aggiunta di un'interfaccia admin sia facilitata.

7.4 Persistenza

I principali componenti del modulo di persistenza sono:

Componente	Funzionalità
StorageService	EJB Singleton che contiene un riferimento all'Entity Manager e che si occupa di realizzare il punto di accesso unico dell'applicazione al database. Tutti i componenti che hanno bisogno di accedere alla memoria persistente devono necessariamente utilizzare i metodi offerti da StorageService.
User	Entity che rappresenta un utente, ne contiene username, password e ruolo.
PrologConfiguration	Entity che rappresenta la configurazione del server, contiene teoria, lista di goal e configuration.

Tabella 7-7- Componenti del modulo di persistenza

Come detto per la persistenza viene usata la versione di Apache Derby integrata in Payara. È da notare che questo è l'unico punto che intacca la portabilità dell'applicazione, che altrimenti funzionerebbe tranquillamente senza modifiche su ogni Application Server che supporti javaee7. In caso di trasferimento su un'altra piattaforma, o se si desidera cambiare il database utilizzato, sarà

sufficiente modificare in modo appropriato il file `persistence.xml` all'interno dell'archivio di cui fare il deploy.

Il database viene utilizzato per conservare i dati relativi a due entity JPA: `User` (contenente username, password e ruolo) e `PrologConfiguration` (contenente un id numerico, teoria, configurazione e goals list).

L'accesso alle entity è regolato da un EJB `StorageService`, utilizzato da tutti gli altri componenti che vogliono accedere agli utenti o alla configurazione (come spiegato nel capitolo 4.1.3). `StorageService` è dunque l'unico componente in cui è iniettato l'`EntityManager` corrispondente alla persistence unit definita in `persistence.xml`, e dovrebbe essere l'unico accesso al database dell'applicazione.

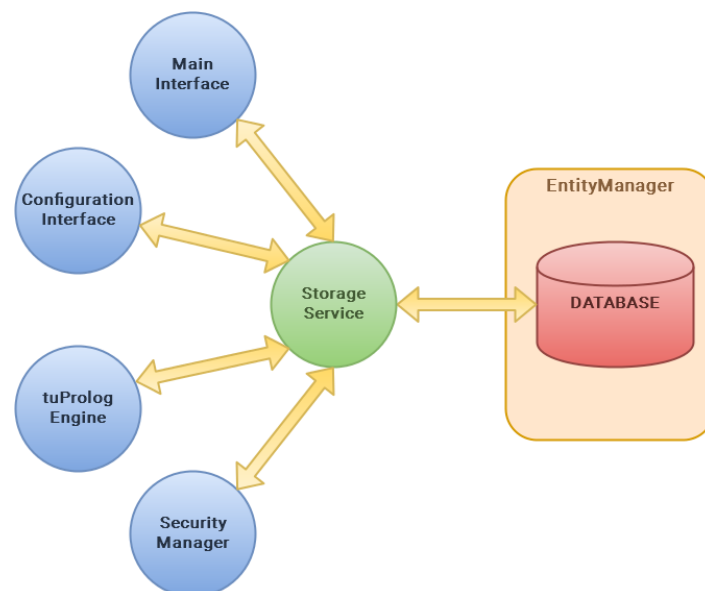


Figura 7-2- Rappresentazione dell'accesso al database. Gli altri moduli possono accedervi solamente tramite `StorageService`. Inoltre `StorageService` stesso non vi accede utilizzando connessioni dirette, ma tramite il layer di astrazione fornito dalle JPA, in particolare tramite l'`EntityManager`.

Fornisce, infatti, metodi per ottenere gli utenti per ruolo, aggiungere utenti, ottenere e resettare la configurazione, eccetera.

7.5 Motore Prolog e concorrenza

Il motore Prolog che costituisce il “core” del servizio è realizzato utilizzando `tuProlog` versione 3.2.0. Il componente `StatelessEngine` funge da wrapper per un oggetto di tipo `Prolog` che utilizza per le varie richieste di risoluzione che riceve. `StatelessEngine` è implementato con uno `Stateless Bean` di cui il container può

fare pooling e gestirne la concorrenza autonomamente e in modo trasparente in caso di richieste contemporanee. Un bean `StatelessEngine` viene iniettato nell'interfaccia principale REST dal container, ed è il mezzo con cui avviene la risoluzione dei goal richiesti dai client.

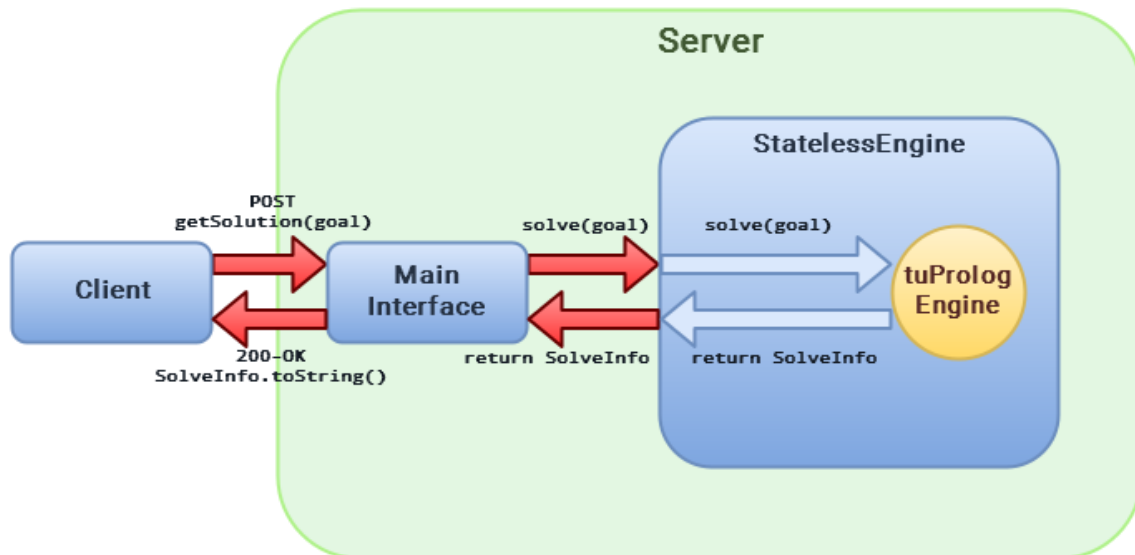


Figura 7-3- Rappresentazione delle chiamate interne al server scatenate da un'invocazione del servizio `getSolution` da parte di un client

`StatelessEngine` supporta inoltre funzioni aggiuntive per la risoluzione con sessione; in questi casi i metodi restituiscono degli oggetti contenitore `PrologSolution`, contenenti una lista di elementi `SolveInfo`, corrispondenti alle varie soluzioni calcolate, e la serializzazione JSON dell'engine alla fine della computazione, ottenuta tramite gli appositi metodi aggiunti nella versione 3.2.0 di `tuProlog`. Alle successive chiamate in una sessione, il metodo di `StatelessEngine`, prima di effettuare la risoluzione caricherà sull'engine lo stato passato come argomento, e quindi riprenderà l'esecuzione da dove era rimasto all'ultima chiamata.

Ogni `StatelessEngine` possiede inoltre un riferimento ad un EJB Singleton di tipo `EngineState`. Questo rappresenta la parte comune a tutti gli `StatelessEngine`, ed è proprio usando `EngineState` che andrebbero apportate le modifiche allo stato interno del motore Prolog, ad esempio tramite caricamenti di librerie, asserzioni o altro, per far in modo che poi vengano propagate a tutti gli `StatelessEngine` e che le sessioni in corso vengano invalidate.

Infatti in tutti i metodi di StatelessEngine, per prima cosa si sincronizza lo stato del bean con quello di EngineState.

In questa versione iniziale del servizio, a fini di test, la sincronizzazione dello stato è limitata alle funzionalità necessarie al test stesso.

In particolare EngineState invia una richiesta ogni minuto ad un componente DataSupplier che si occupa di generare casualmente dei dati (si veda a tal proposito il capitolo successivo (7.6) o quello sullo scenario applicativo(**Errore. L'origine riferimento non è stata trovata.**)). A questo punto EngineState genera dei termini Prolog a partire dai dati e li inserisce in una lista; quando uno StatelessEngine tenterà di sincronizzarsi, effettuerà il retract dei termini correntemente asseriti, otterrà la lista aggiornata da EngineState e infine effettuerà l'assert dei nuovi termini.

7.6 Caricamento dati

Per ampliare le possibilità e i campi d'uso dell'applicazione server è auspicabile che essa sia in grado di ricevere dati in qualche modo, o da fonti esterne o tramite componenti interni.

È dunque necessario realizzare un componente EJB DataSupplier che, leggendo un file di configurazione data_configuration.xml nell'archivio di deploy, generi casualmente (almeno in questa versione prototipale dell'applicazione) dei dati basandosi sulle informazioni in esso contenute.

Questo file XML contiene un tag di root chiamato <data> che può contenere svariati elementi <element>, ciascuno corrispondente ad un diverso dato da generare.

Ogni element contiene al suo interno due tag obbligatori <name> e <type>; il primo contiene il nome che identifica il dato, il secondo il suo tipo (a scelta tra “int”, “double” e “boolean”). Infine, solo in caso il tipo del dato sia int o double, element contiene un terzo tag chiamato <range> che a sua volta contiene due tag: <from> e <to>. Questi due ultimi tag contengono i valori, rispettivamente minimo e massimo, accettati per la generazione casuale di quel dato.

Ad esempio un possibile file `data_configuration.xml` potrebbe presentarsi come segue:

```
<data>
  <element>
    <name>test_int</name>
    <type>int</type>
    <range>
      <from>15</from>
      <to>46</to>
    </range>
  </element>
  <element>
    <name>test_double</name>
    <type>double</type>
    <range>
      <from>10.67</from>
      <to>67.43</to>
    </range>
  </element>
  <element>
    <name>test_boolean</name>
    <type>boolean</type>
  </element>
</data>
```

Codice 7-1- Esempio di file `data_configuration.xml` contenente tre elementi da generare casualmente: uno di nome `test_int` di tipo `int` e con valori casuali da 15 a 46, `test_double` di tipo `double` e con valori casuali da 10.67 a 67.43, e un ultimo di nome `test_boolean` e tipo `boolean` (ovviamente è inutile specificare un range di valori).

I dati generati vengono restituiti dal metodo `getData()` sotto forma di Lista di `JsonObject`, ciascuno con tre proprietà:

- **“name”**: il nome associato al valore generato, estratto dal contenuto del tag `<name>`.
- **“type”**: il tipo del valore generato, estratto dal contenuto del tag `<type>`.
- **“value”**: il valore generato casualmente.

Come già accennato, questo metodo al momento viene chiamato dal componente `EngineState` ogni minuto per simulare il caricamento di nuovi dati. Sebbene questo comportamento di tipo pull per ottenere i dati sia plausibile anche in una versione finale dell'applicazione, è chiaro che il funzionamento del metodo `getData()` di `DataSupplier` andrebbe completamente rivisto per ottenere dati reali, da un EJB interno o esterno, o magari da un altro Web Service online, o da una pagina web e così via.

Inoltre è da notare come non sempre un comportamento di tipo pull per ottenere i dati sia efficace. Nel caso, ad esempio, di un dispositivo esterno che genera dati in momenti non prevedibili a priori, la modalità più naturale è che esso stesso li invii al server in modo push. Affinché questo sia possibile sarebbe necessario lo sviluppo di un'ulteriore interfaccia (realizzabile tramite WS RESTful, EJB, o un approccio ibrido come già mostrato), al quale il dispositivo possa mandare i dati; questa interfaccia tuttavia non è stata realizzata nell'ambito di questa tesi, poiché non immediatamente necessaria ai fini del test di funzionamento.

7.7 Applicazioni Client

Per realizzare le applicazioni client può essere utilizzata una libreria che realizzi un client http come ad esempio Apache httpClient, per la ricezione e l'invio di messaggi da e verso il server, una per la serializzazione/deserializzazione JSON come la già discussa Google gson e una per la navigazione dei dati in XML come ad esempio JDOM.

Anche per le applicazioni client c'è margine di ampliamento. Per semplicità, ad esempio, viene utilizzata la modalità sincrona di httpClient per la comunicazione con il server che tuttavia potrebbe essere integrata anche con una modalità asincrona.

8. Implementazione: un caso concreto

8.1 Illustrazione dello scenario applicativo

Si immagini un ipotetico bagno “smart” in un’ottica IoT; il bagno in questione sarebbe costituito da una serie di componenti molto avanzati, costantemente connessi ad internet e che possono fare uso di sensori avanzati, in grado di estrarre dati significativi per l’utente dalle semplici azioni che egli svolge ogni giorno. Si pensi, ad esempio, uno spazzolino che possa raccogliere dati sulla presenza di batteri pericolosi nella saliva, in modo simile ad un tampone, o un gabinetto che possa rilevare glucosio nelle urine o sangue occulto nelle feci, come avviene per i relativi esami in ospedale. Molte di queste tecnologie non sono disponibili, tuttavia la direzione generale del mercato è quella di dispositivi sempre più connessi e sempre più “intelligenti”, non è pertanto assurdo pensare ad un futuro prossimo in cui dispositivi del genere siano realtà.

Questi elementi del bagno smart tuttavia non sono semplici sensori, ma possono possedere anche una unità di elaborazione associata, un processore a consumo ridotto in grado di elaborare, almeno parzialmente, questi dati. Si pensi ad esempio ad un dispositivo paragonabile ad un Raspberry Pi, o ad un suo equivalente. È proprio su questo tipo di unità che dovrebbe eseguire l’applicazione precedentemente trattata.

Il server si occuperebbe di ricevere i dati dal sensore, inserirli nel suo motore inferenziale tuProlog dove avverrebbe una elaborazione iniziale, e tramite le interfacce web, sarebbe quindi interrogabile da qualsiasi client ne conosca l’indirizzo. Allo stesso modo sarebbe anche configurabile totalmente in remoto, utilizzando l’apposita interfaccia di configurazione.

In questo scenario si potrebbe avere infine un dispositivo controllato in modo diretto dall’utente (ad esempio uno smartphone, un tablet o anche uno smartwatch), anch’esso fornito di un motore Prolog, ma non necessariamente dell’applicazione server, che funzioni da collettore di dati e che si occupi di interrogare i vari

componenti del bagno smart. Questo dispositivo una volta raccolti i dati potrebbe elaborarli ulteriormente tramite il suo motore d'inferenza, combinarli e confrontarli a seconda delle sue regole interne, ed elaborarne, ad esempio, una lista di segnalazioni o avvertenze con diversi livelli di priorità.

Usando questo scenario come punto di riferimento si può realizzare una versione in miniatura e simulata dello stesso, utilizzando un numero limitato di elementi e utilizzando dati generati casualmente.



Figura 8-1- Un ipotetico bagno smart con tre elementi: un gabinetto, uno spazzolino e un dispositivo wearable, che offrono servizi evoluti grazie all'applicazione server realizzata e che possono essere interrogati da un dispositivo consumer come un tablet o uno smartphone per ricevere notifiche di alto livello elaborate a partire da dati raw.

Ad esempio possiamo limitare gli elementi ai seguenti:

- **Gabinetto:** effettua l'esame delle urine e delle feci estraendone dati sulla salute dell'utente. Riceve dati come la presenza di glucosio o globuli bianchi nelle urine, il livello di sodiuria, la presenza di sangue occulto nelle feci ed elabora una lista di sintomi (disidratazione, sodiuria elevata, ecc...) e malattie (possibilità di diabete o prediabete).
- **Spazzolino elettrico:** genera una serie di dati sul funzionamento dello spazzolino stesso e ricerca la presenza di batteri nella saliva. Può segnalare

una possibile infezione, o dare avvertenze generiche come la necessità di lavarsi i denti se son passate troppe ore dall'ultima attivazione.

- **Dispositivo Personale:** un ipotetico dispositivo personale indossabile, in grado di rilevare i segni vitali principali dell'utente, come temperatura, pressione sistolica e diastolica e battito cardiaco. Anche questo ovviamente svolge le sue elaborazioni interne quando viene interrogato e fornisce possibili sintomi (tachicardia, temperatura elevata, pressione alta) o malattie (febbre).

Infine, come dispositivo collettore, è sufficiente un generico tablet o smartphone. Si noti che configurando correttamente il router al quale i dispositivi smart sono collegati (in particolare collegando IP locale e porta del dispositivo ad una porta del router), i vari dispositivi non sono più raggiungibili esclusivamente dalla stessa rete locale, ma diventano consultabili anche da qualsiasi dispositivo tramite internet. Questa possibilità, sebbene pienamente funzionante, per essere realisticamente utilizzata necessiterebbe prima dell'implementazione delle già menzionate funzionalità di gestione degli utenti da parte di un admin, in modo da permettere l'accesso online solamente a determinati utenti.

8.2 Realizzazione hardware

Per la realizzazione pratica è ovviamente opportuno avvicinarsi quanto più possibile ad una implementazione realistica dello scenario di test, utilizzando hardware diversificati sia a livello di sistema operativo e architettura hardware, sia a livello di prestazioni, per simulare una situazione reale con dispositivi eterogenei, estremamente diversi l'uno dall'altro. Uno spazzolino o un dispositivo indossabile avranno, anche solamente per necessità di dimensioni, un hardware e delle prestazioni decisamente diverse da quelle di un gabinetto, che magari può permettersi di avere integrato un hardware più ingombrante e performante.

Per la simulazione del bagno "smart" sono quindi utilizzati i seguenti tre dispositivi con le rispettive specifiche:

Dispositivo Simulato	Dispositivo Reale	Architettura	OS	Caratteristiche Tecniche	Java version
Gabinetto	Personal Computer	Intel x64	Windows 10 x64	Intel Core i7-6700 @3.40GHz / 8GB RAM DDR4	1.8.0_u122
Spazzolino	Acer Aspire One D270 (Netbook)	Intel x64	Ubuntu 15.10 32bit	Intel Atom N2600 @1.60GHz / 3GB RAM DDR3	1.8.0_u91
Wearable Device	Raspberry Pi 3	ARMv8 (64bit)	Ubuntu Mate 16.04	Broadcom BCM2837 @1.2GHz / 1GB RAM LPDDR2	1.8.0_u111 (ARM)

Si può notare come si spazi tra diverse architetture hardware (Intel e ARM), diversi livelli di performance (da potenti processori per PC di fascia alta, a piccole CPU mirate al risparmio energetico), e diversi Sistemi Operativi.

In questo scenario diventa particolarmente importante vedere come si comporta l'applicazione server con a disposizione solo risorse molto limitate, come nel caso del netbook e del Raspberry Pi.



Figura 8-2- Scenario applicativo aggiornato con i dispositivi utilizzati per la simulazione dei vari elementi del bagno smart

Tutti e tre i dispositivi sono collegati in rete tramite un normale router wireless (TP-LINK Archer D2), il Raspberry e il netbook utilizzando il Wi-Fi, il PC utilizzando un collegamento via cavo.

Il dispositivo personale, con il quale consultare i risultati, è implementato da un comune tablet Android, in particolare un Lenovo TAB 2 A10 che fa girare Android in versione 5.0.1, anch'esso collegato via Wi-Fi allo stesso router.

Per completezza viene anche effettuato un test, come precedentemente descritto, effettuando la connessione ai singoli dispositivi via internet, connettendo il tablet ad una rete terza.

8.3 Realizzazione Software

8.3.1 Configurazione

Tutti e tre i dispositivi server montano la stessa applicazione discussa nei capitoli precedenti. Le personalizzazioni, per specificarne il diverso comportamento, avvengono in due modi: la prima tramite configurazione e relativa interfaccia, la seconda tramite file di configurazione.

È possibile, e necessario, specificare la teoria con la quale il servizio dovrà lavorare ed elaborare le soluzioni richieste, e la lista di goal di cui sarà possibile richiedere la soluzione. Questo tipo di configurazione può avvenire a runtime utilizzando l'interfaccia REST, una volta che il server è già in funzione, mentre il tipo di dati generati per il test può essere configurato solo a deploy time, modificando il file `data_configuration.xml` all'interno dell'archivio war.

Il file XML contiene, per ciascun servizio, tutti i dati che il rispettivo sensore simulato genererà casualmente ogni minuto. Si noti che questo file è utilizzato esclusivamente a fini di test, e la sua stessa esistenza non avrebbe senso in una versione finale dell'applicazione, dove i dati non vengono generati casualmente a partire da un template, ma dovrebbero essere ottenuti da qualche altra fonte.

La configurazione tramite interfaccia dovrà impostare la teoria e i goal che funzioneranno sui dati generati casualmente, e in realtà, non è necessario ripeterla ad ogni avvio del server, poiché è salvata in memoria persistente.

Per semplificare e velocizzare le cose è possibile realizzare un piccolo programma in Java, da lanciare una sola volta, che si occupi di effettuare la prima configurazione dei tre servizi con i dati necessari.

Per i dettagli sulle configurazioni usate si veda il capitolo 10.1.

8.3.2 Application Server

Sulle rispettive macchine l'applicazione gira su Application Server Payara versione 4.1.1.164, tuttavia su Raspberry Pi e netbook viene utilizzata la già discussa versione Micro di Payara, una versione pensata esclusivamente per server e servizi leggeri, senza la necessità di installare l'intero Application Server. Payara Micro è distribuito come semplice file .jar, eseguibile da una comune JVM. Il deploy e la configurazione di applicazioni può avvenire in modo programmatico, utilizzando Payara Micro in un normale programma Java, o semplicemente da linea di comando. In questo modo tramite un solo comando e una normale JVM è possibile effettuare il deploy, perfettamente funzionante, dell'applicazione.

In particolare si è utilizzato:

```
java -jar payara-micro.jar --rootDir ./payaraRoot  
--deploy rest.war --noCluster
```

Codice 8-1- Comando per l'esecuzione dell'applicazione su Payara Micro

- **rootDir:** questa opzione permette di specificare la cartella di root utilizzata da Payara Micro, qui verranno inserite o recuperate le impostazioni dell'application server, ed è sempre qui che verranno persistiti i dati in caso si utilizzi il database Apache Derby integrato. A default, se non specificato, la cartella verrà creata in temp. Nel nostro caso è particolarmente importante specificare una cartella di root, affinché i dati persistiti su database vengano ritrovati ad un avvio successivo.

- **deploy:** con questa opzione si specifica l'archivio di cui fare il deploy. È anche disponibile un comando `--deploymentDir`, che permette di effettuare il deploy di tutti gli archivi contenuti nella cartella specificata.
- **noCluster:** Payara Micro è integrato con Hazelcast e offre un supporto esteso al clustering, tanto che questo è abilitato a default. Eseguendo il comando senza questa opzione i servizi si riconoscerebbero automaticamente tra di loro e creerebbero un cluster, soluzione che, sebbene sia interessante per possibili studi futuri, in questo test rende particolarmente complicato verificare il funzionamento delle singole applicazioni.

Payara Micro permette di specificare anche numerose altre opzioni, per configurare il clustering in modo più approfondito, per cambiare la porta a cui risponde il server e così via. Una menzione particolare va fatta per l'opzione `outputUberJar` che permette di impacchettare il server e le applicazioni di cui si è fatto il deploy in un unico “uber” file jar. Eseguendolo si avvierà il server con le impostazioni configurate e con le applicazioni di cui si è stato specificato il deploy.

8.3.3 Applicazione Android

Infine, per implementare la funzionalità di client dei vari servizi del bagno smart, si può realizzare una applicazione Android. L'applicazione in questione è stata realizzata in Java utilizzando Android Studio e l'Android SDK.

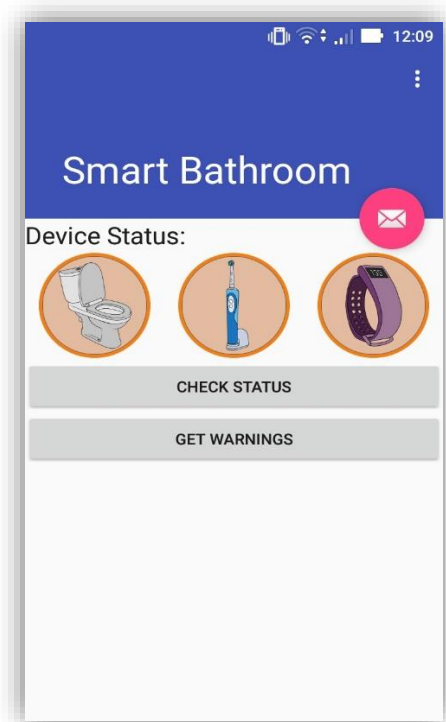


Figura 8-3- Interfaccia grafica della applicazione Android

L'interfaccia grafica è abbastanza spartana, ma contiene tutti gli elementi fondamentali. Le tre icone rappresentano i tre elementi del nostro bagno smart e, in particolare, il colore del loro sfondo indica l'attuale stato di raggiungibilità di quel server. Alla pressione del pulsante "CHECK STATUS" l'applicazione tenta di contattare i tre server e, se ottiene risposta, lo sfondo delle rispettive icone viene colorato di verde, in caso contrario lo sfondo diventa arancione (come in Figura 8-3).

Premendo invece il pulsante "GET WARNINGS", l'applicazione interrogherà i tre server e otterrà gli avvisi corrispondenti che poi verranno visualizzati a schermo codificati per colore (avvisi urgenti con sfondo arancione, avvisi non urgenti con sfondo verde).

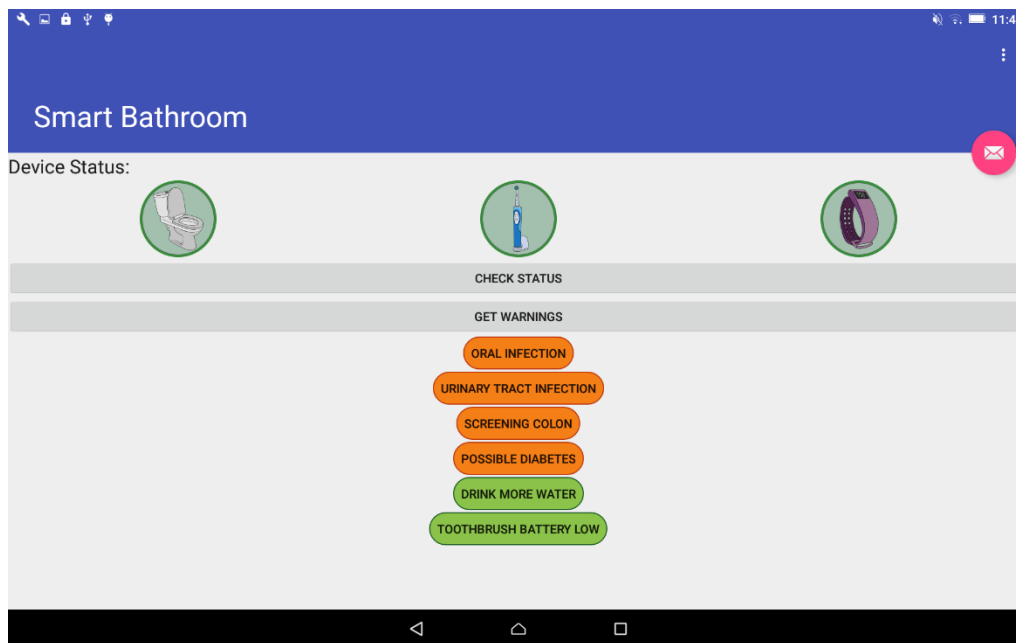


Figura 8-4- Visualizzazione a schermo degli avvisi urgenti (arancioni) e non (verdi), in seguito alla pressione del pulsante “GET WARNINGS”

L'applicazione permette di specificare gli indirizzi a cui trovare i server che simulano i vari componenti del bagno smart tramite una pagina “Settings” raggiungibile tramite il menu.

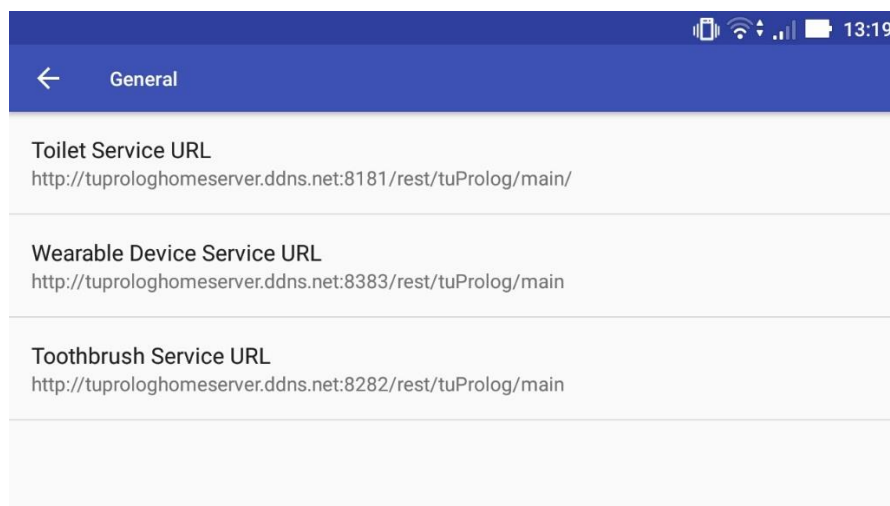


Figura 8-5- Schermata delle impostazioni dell'applicazione. Permette di specificare gli URL a cui trovare i server del bagno smart.

L'applicazione contiene al suo interno un engine tuProlog, configurato con apposita teoria (si veda capitolo 10.1) per elaborare avvisi urgenti e non, a partire dai dati ricevuti dai tre dispositivi smart.

Al click sul pulsante “GET WARNINGS”, la `MainActivity` resetta le liste dei risultati e infine avvia tre task asincroni, uno per ogni dispositivo da interrogare. Ciascuno di questi task chiede la risoluzione dei goal disponibili sul dispositivo tramite richiesta http (viene utilizzata l’implementazione di client http `okhttp`) e inserisce le soluzioni ottenute in una lista. L’ultimo task a completare l’esecuzione, elabora la lista dei risultati generando delle asserzioni per aggiornare la knowledge base del motore `tuProlog` interno e infine lo interroga per ottenere due liste di avvisi divisi per urgenza, con le quali aggiorna la GUI.

Per come è strutturata l’intera simulazione, le soluzioni cambiano ogni minuto, quando i dati dei servizi vengono rigenerati.

Per un esame passo a passo dei dati in una interazione di esempio si veda l’appendice 10.2.

8.4 Collaudo e risultati

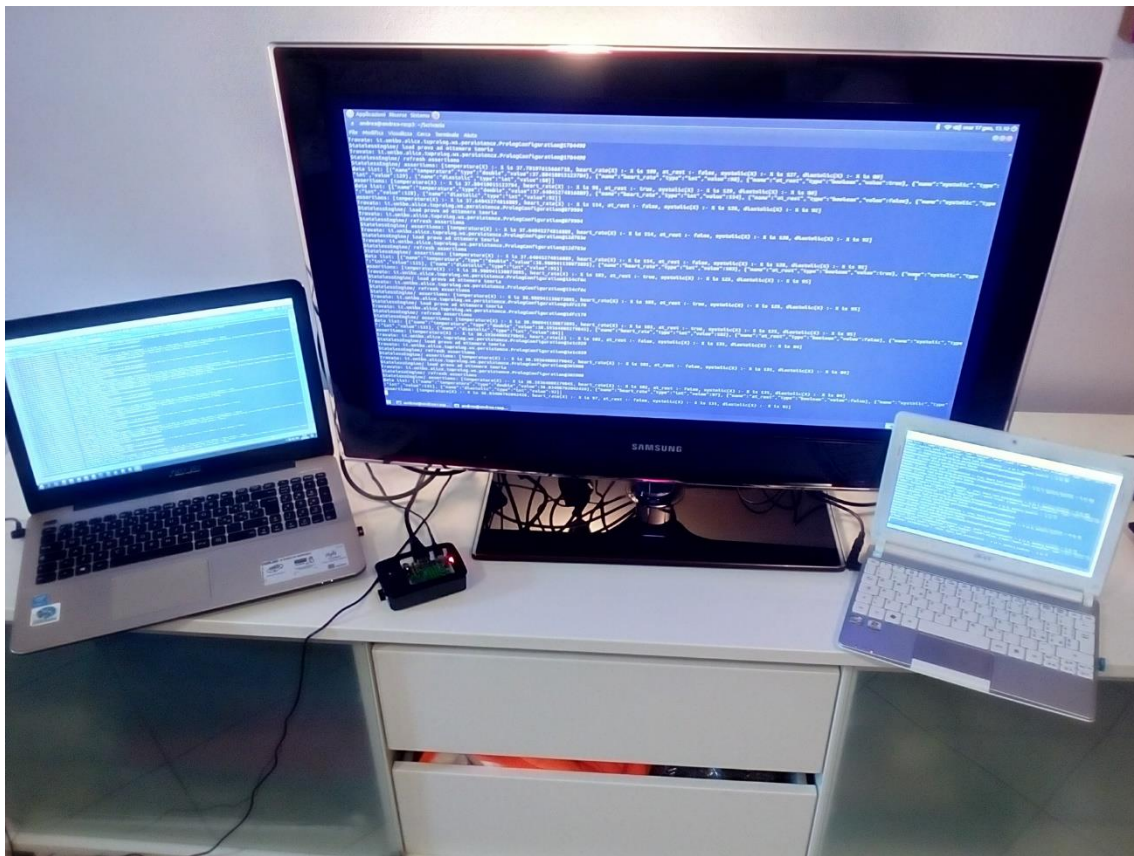


Figura 8-6- Setup dei dispositivi per la simulazione. Sulla destra il netbook che simula lo spazzolino, al centro lo schermo a cui è collegato il Raspberry Pi che simula il dispositivo indossabile, e a sinistra un portatile che visualizza lo schermo del PC su cui è simulato il gabinetto.

A questo punto tutto ciò che serve ai fini della simulazione dello scenario applicativo è pronto ed è solo necessario eseguire le dovute operazioni in ordine:

1. (Opzionale) Se non si usa Payara Micro, avviare il database con il comando `start-database` dalla console `asadmin` di Payara.
2. Avviare gli Application Server sulle relative macchine.
3. Effettuare il deploy degli archivi `.war` sui rispettivi AS (questo punto e il precedente avvengono insieme nel caso di Payara Micro)
4. (Opzionale) Se è il primo avvio in assoluto, eseguire la configurazione delle applicazioni.
5. Avviare l'applicazione Android e ottenere le liste di avvisi.

L'avvio e il deploy avviene con successo anche sui dispositivi meno potenti (Raspberry Pi e netbook), dipendentemente dai casi il bootstrap completo prende

dai 40 ai 60 secondi su entrambi i dispositivi. Bisogna ricordare tuttavia che in questi due casi si utilizza Payara Micro, e la procedura d'avvio comprende non solo il deploy e l'avvio dell'applicazione (che rappresenta una minima percentuale del tempo totale), ma anche, e soprattutto, l'avvio dell'intero Application Server. Su PC l'avvio dell'Application Server prende tra i 10 e i 15 secondi, mentre il deploy dell'applicazione è praticamente istantaneo.

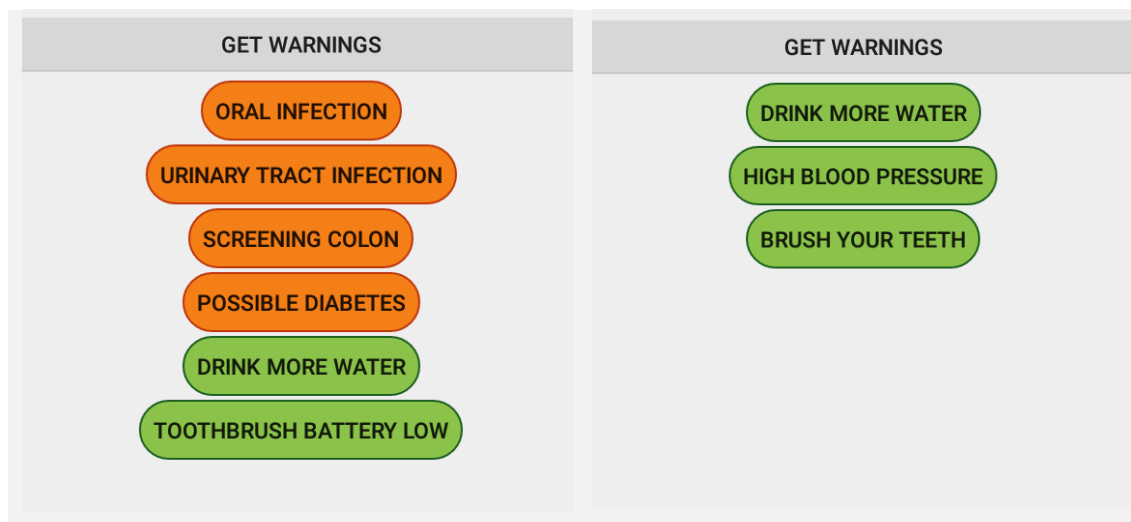
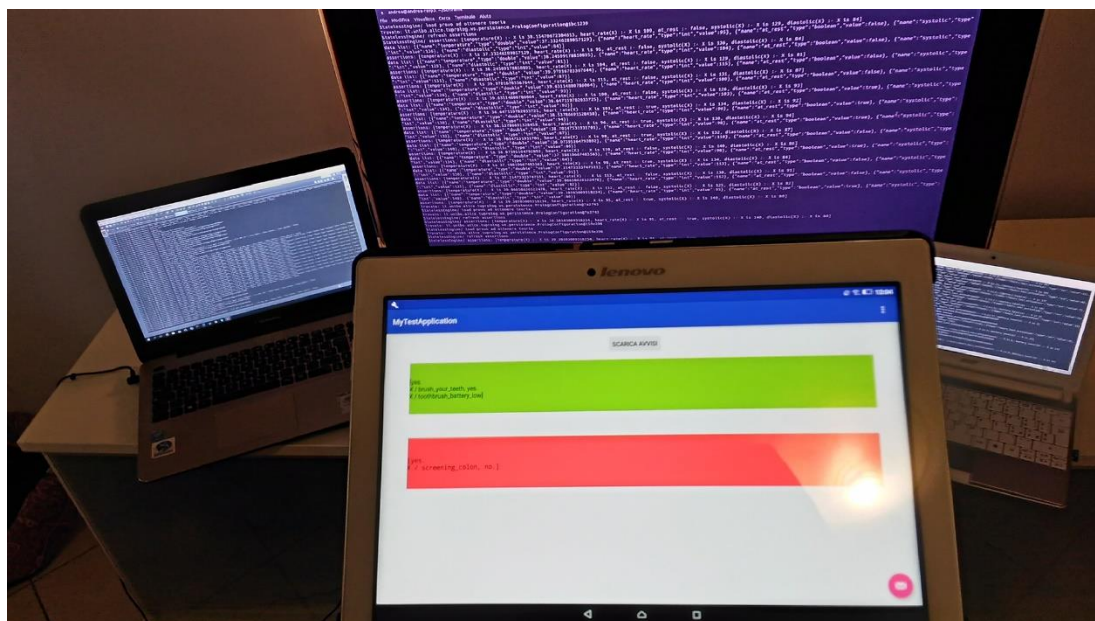


Figura 8-7- Sopra è mostrata una versione iniziale dell'applicazione Android in esecuzione assieme al resto dei dispositivi server. Sotto si possono notare degli screenshot che mostrano due differenti risultati visualizzati sull'applicazione Android, elaborati a partire da set di dati diversi.

Una volta avviati, a schermo iniziano a comparire di minuto in minuto i dati generati casualmente, in modo da poter verificare che le liste generate dall'applicazione Android siano corrette. Premendo il pulsante “GET

WARNINGS” nell’app, compaiono i risultati e si nota come siano effettivamente corretti e basati sui dati attualmente resi disponibili dai server. Il tempo impiegato dalla pressione del pulsante all’apparizione a schermo dei risultati è quasi istantaneo e non sembrano esserci rallentamenti di sorta dovuti all’impiego di hardware poco performante.

In definitiva questo test dimostra il corretto funzionamento e le potenzialità dell’applicazione realizzata, mettendola a confronto con un caso d’uso realistico.

9. Conclusioni

Quanto realizzato è, come detto più volte, un prototipo e sebbene sia completamente funzionante, durante il suo sviluppo si è data minore importanza ad elementi non strettamente necessari alla realizzazione delle funzionalità principali richieste; per queste ragioni ci sono ampi margini di sviluppo e miglioramento.

Una possibile estensione potrebbe riguardare il miglioramento del modulo di sicurezza; ad esempio fornendo una interfaccia di amministrazione che consenta la gestione remota e dinamica, piuttosto che da configurazione statica, dei ruoli, degli utenti e delle opzioni di accesso.

Un'altra possibile estensione riguarda la gestione dei dati; si potrebbero realizzare metodi alternativi per il loro ottenimento come, ad esempio, un'interfaccia per riceverli in modo push da altri dispositivi, o ancora testare il prototipo con dati reali provenienti in real-time da sensori piuttosto che con dati di test generati casualmente.

Infine è possibile esplorare parti non prese in esame da questi tesi, come la possibilità di implementare un sistema di caching dei risultati per migliorare le performance lato server, o ancora esaminare le funzionalità di clustering offerte da Payara Micro.

Durante questa tesi si è dunque analizzato il modello LPaaS, e si è studiata una possibile implementazione dello stesso, seguendone lo sviluppo in tutte le fasi.

Si è costruita una possibile struttura architetturale del software e sono state analizzate le tecnologie attualmente disponibili per implementarla. Sono quindi state usate le suddette tecnologie per arrivare alla realizzazione di un prototipo funzionante.

Durante lo svolgimento sono emerse diverse problematiche quali la gestione dello stato di sessione in un ambiente stateless, o l'interazione tra accesso concorrente

ad una knowledge base condivisa, alle quali tuttavia sono state fornite soluzioni efficaci sia in fase progettuale che implementativa.

Si è infine dimostrata l'effettiva funzionalità del prototipo realizzato utilizzandolo all'interno di una simulazione di uno scenario applicativo che potrebbe realisticamente presentarsi in un futuro non troppo remoto.

10. Appendici

10.1 Configurazioni e dati per il caso d'uso

Come detto i tre diversi server utilizzati per il test sono identici se si esclude la diversa configurazione. Parte di questa configurazione avviene già a deploy time, con il file `data_configuration.xml`, che specifica i dati da generare casualmente ai fini del test (per informazione su come interpretare questi file si veda il capitolo 7.6)

1. In particolare sono stati usati i seguenti file xml:

```
<!-- TOILET / DATA_CONFIGURATION.XML -->
<data>
  <element>
    <name>specific_weight</name>
    <type>int</type>
    <range>
      <from>1026</from>
      <to>1035</to>
    </range>
  </element>
  <element>
    <name>urine_sodium</name>
    <type>int</type>
    <range>
      <from>197</from>
      <to>205</to>
    </range>
  </element>
  <element>
    <name>wbc</name>
    <type>boolean</type>
  </element>
  <element>
    <name>blood</name>
    <type>boolean</type>
  </element>
  <element>
    <name>ketones</name>
    <type>boolean</type>
  </element>
  <element>
    <name>glucose</name>
    <type>boolean</type>
  </element>
</data>
```

Codice 10-1- data_configuration.xml utilizzato per generare i dati casuali sul dispositivo che simula il gabinetto all'interno dello scenario applicativo di un bagno smart

In cui i valori rappresentano, in ordine, il peso specifico delle urine, la sodiuria delle urine, la presenza di globuli bianchi nelle urine, la presenza di sangue nelle feci, la presenza di chetoni nelle urine e la presenza di glucosio nelle urine.

```
<!-- TOOTHBRUSH / DATA_CONFIGURATION.XML -->
<data>
  <element>
    <name>streptococcus</name>
    <type>boolean</type>
  </element>
  <element>
    <name>hours_last_activation</name>
    <type>int</type>
    <range>
      <from>6</from>
      <to>12</to>
    </range>
  </element>
  <element>
    <name>battery_level</name>
    <type>int</type>
    <range>
      <from>2</from>
      <to>25</to>
    </range>
  </element>
</data>
```

Codice 10-2- data_configuration.xml utilizzato per generare i dati casuali sul dispositivo che simula lo spazzolino all'interno dello scenario applicativo di un bagno smart

I valori rappresentano, in ordine, la presenza o meno di streptococco nella saliva, le ore passate dall'ultima attivazione e il livello percentuale della batteria dello spazzolino.

```

<!-- WEARABLE / DATA_CONFIGURATION.XML -->
<data>
  <element>
    <name>temperature</name>
    <type>double</type>
    <range>
      <from>36</from>
      <to>40</to>
    </range>
  </element>
  <element>
    <name>heart_rate</name>
    <type>int</type>
    <range>
      <from>95</from>
      <to>115</to>
    </range>
  </element>
  <element>
    <name>at_rest</name>
    <type>boolean</type>
  </element>
  <element>
    <name>systolic</name>
    <type>int</type>
    <range>
      <from>125</from>
      <to>140</to>
    </range>
  </element>
  <element>
    <name>diastolic</name>
    <type>int</type>
    <range>
      <from>80</from>
      <to>95</to>
    </range>
  </element>
</data>

```

Codice 10-3- data_configuration.xml utilizzato per generare i dati casuali sul dispositivo che simula il dispositivo wearable all'interno dello scenario applicativo di un bagno smart

I valori rappresentano, in ordine, la temperatura corporea, il battito cardiaco, se si è o meno a riposo, la pressione sistolica e quella diastolica.

Si noti che i valori dei range di int e double sono stati impostati in modo da favorire un esito positivo del test, piuttosto che una rappresentazione realistica.

Il resto della configurazione avviene a runtime tramite il servizio RESTful di configurazione. Qui si impostano teoria degli engine Prolog e la lista di goal a disposizione degli utenti.

La knowledge base dei server viene aggiornata di volta in volta in modo controllato dal server stesso, come già spiegato precedentemente, ricavando comandi di assert a partire dai dati generati.

Per il server che simula il gabinetto è stata utilizzata la seguente teoria:

```
disease(prediabetes) :- glucose, not(disease(diabetes)).
disease(diabetes) :- glucose, ketones.
symptom(high_urine_sodium) :- urine_sodium(X), X > 200.
symptom(dehydration) :- specific_weight(X), X > 1028.
symptom(urine_wbc) :- wbc.
symptom(occult_blood) :- blood.
```

Codice 10-4- Teoria utilizzata per il motore Prolog che simula il gabinetto

Mentre i due goal disponibili nella goal list sono:

- **symptom(X).**
- **disease(X).**

Per il server che simula lo spazzolino è stata utilizzata la seguente teoria:

```
symptom(infection) :- streptococcus.
warning(brush_your_teeth) :- hours_last_activation(X), X > 8.
warning(battery_low) :- battery_level(X), X < 16.
```

Codice 10-5- Teoria utilizzata per il motore Prolog che simula lo spazzolino

Mentre i due goal disponibili nella goal list sono:

- **symptom(X).**
- **warning(X).**

Per il server che simula il wearable device è stata utilizzata la seguente teoria:

```
symptom(high_temperature) :- temperature(X), X >= 38.
symptom(tachycardia) :- heart_rate(X), X > 100, at_rest.
symptom(high_blood_pressure) :- systolic(X), diastolic(Y),
                                X > 130, Y > 85, at_rest.
symptom(low_blood_pressure) :- systolic(X), diastolic(Y),
                                X < 110, Y < 75, at_rest.
disease(fever) :- symptom(high_temperature), symptom(tachycardia).
```

Codice 10-6- Teoria utilizzata per il motore Prolog che simula il dispositivo wearable

Mentre i due goal disponibili nella goal list sono:

- **symptom(X).**
- **disease(X).**

Infine viene utilizzato un ultimo engine tuProlog, all'interno dell'applicazione Android, per elaborare e combinare i dati ricevuti risolvendo i goal di tutti e tre i

servizi e ottenere due liste di avvisi con diversa priorità. Quest'ultimo motore Prolog utilizza la seguente teoria:

```
warning1(limit_sodium_intake) :- high_urine_sodium,  
                                high_blood_pressure.  
warning1(drink_more_water) :- dehydration.  
warning1(high_blood_pressure) :- high_blood_pressure.  
warning1(low_blood_pressure) :- low_blood_pressure.  
warning1(brush_your_teeth) :- brush_your_teeth.  
warning1(toothbrush_battery_low) :- battery_low.  
warning2(oral_infection) :- fever, infection.  
warning2(urinary_tract_infection) :- fever, urine_wbc.  
warning2(screening_colon) :- occult_blood.  
warning2(possible_prediabetes) :- prediabetes.  
warning2(possible_diabetes) :- diabetes.
```

Codice 10-7- Teoria utilizzata per il motore Prolog all'interno dell'applicazione Android che si occupa di generare due liste di avvisi a diversa priorità a partire dai risultati ottenuti dagli elementi del bagno smart

Mentre i due goal utilizzati sono:

- **warning1(X).**
- **warning2(X).**

10.2 Dai dati generati alle liste di avvisi: un esempio

Di seguito viene esaminato il percorso dei dati all'interno dello scenario di test precedentemente descritto.

Inizialmente i dati vengono generati casualmente a partire dalle informazioni contenute nei file `data_configuration.xml`. I file usati nel test sono mostrati nell'appendice precedente. Responsabile della generazione di questi dati è il componente `DataSupplier` dell'applicazione server; il suo metodo `getData()`, restituisce una lista di `JsonObject` contenente i dati random.

Supponiamo che i vari servizi generino i seguenti dati:

TOILET SERVICE- specific_weight : 1031 urine_sodium : 202 wbc : false blood : false ketones : false glucose : true	WEARABLE DEVICE SERVICE- temperature : 38.45 heart_rate : 106 at_rest : true systolic : 134 diastolic : 87
TOOTHBRUSH SERVICE- streptococcus : true hours_last_activation : 10 battery_level : 24	

Codice 10-8- Dati raw generati casualmente dai vari servizi del bagno smart

A questo punto il componente `EngineState`, dopo aver invocato `getData()`, elabora la lista di `JsonObject` ricevuta per generare una lista di termini da asserire per aggiornare la relativa knowledge base.

Ad esempio dai dati mostrati sopra verrebbero generati i seguenti termini:

TOILET SERVICE- specific_weight(X) :- X is 1031 urine_sodium(X) :- X is 202 wbc :- false blood :- false ketones :- false glucose :- true	WEARABLE DEVICE SERVICE- temperature(X) :- X is 38.45 heart_rate(X) :- X is 106 at_rest :- true systolic(X) :- X is 134 diastolic(X) :- X is 87
TOOTHBRUSH SERVICE- streptococcus :- true hours_last_activation(X) :- X is 10 battery_level(X) :- X is 24	

Codice 10-9- Conversione in termini da asserire dei dati raw casuali

Ogni volta che uno StatelessEngine tenterà di sincronizzarsi, effettuerà il retract dei termini correntemente asseriti ed eseguirà l'assert dei termini aggiornati contenenti i nuovi dati.

Quando il client invocherà la risoluzione dei goal sui servizi otterrà i seguenti risultati, basati sui dati generati e sulle teorie configurate (si veda l'appendice precedente):

TOILET SERVICE- symptom(X) . X / high_urine_sodium X / dehydration	WEARABLE DEVICE SERVICE- symptom(X) . X / high_temperature X / tachycardia X / high_blood_pressure
TOILET SERVICE- disease(X) . X / prediabetes	WEARABLE DEVICE SERVICE- disease(X) . X / fever
TOOTHBRUSH SERVICE- symptom(X) . X / infection	
TOOTHBRUSH SERVICE- warning(X) . X / brush_your_teeth	

Codice 10-10- Risultati ottenuti risolvendo i rispettivi goal sui vari dispositivi che realizzano il bagno smart

In modo del tutto simile da quanto fatto dai servizi precedentemente, anche il client genererà dei termini da asserire da questi risultati e aggiornerà la knowledge base del proprio motore Prolog.

A questo punto il client interrogherà il proprio engine per ottenere le liste di avvertenze (una urgente e una no) da restituire al client:

CLIENT- warning1(X) . X / limit_sodium_intake X / drink_more_water X / high_blood_pressure X / brush_your_teeth	CLIENT- warning2(X) . X / possibile_prediabetes X / oral_infection
--	---

Codice 10-11- Risultati finali ottenuti dall'applicazione Android risolvendo i rispettivi comandi sul proprio motore Prolog interno

10.3 Guida all'installazione

In questa appendice verrà spiegato passo a passo come arrivare all'esecuzione dell'applicazione partendo dal codice del progetto.

10.3.1 Esecuzione su Payara Server e Eclipse

Questa parte spiegherà come arrivare al deploy ed esecuzione dell'applicazione su una installazione di Eclipse for Java EE^[ECL] (testato con Eclipse Neon 4.6 for Java EE) collegata ad un Application Server Payara (testato con Payara Server Full 4.1.1.164).

Preparazione:

1. Scaricare Payara Server all'indirizzo: <http://www.payara.fish/downloads>
2. Installare Payara estraendolo in una cartella (ad esempio C:\payara41)
3. Installare il plugin Eclipse “GlassFish Tools” per poter aggiungere e configurare server GlassFish o Payara su Eclipse. Il plugin è scaricabile alla pagina: <https://marketplace.eclipse.org/content/glassfish-tools>
4. In Eclipse aggiungere un nuovo server selezionando “File>New>Server”

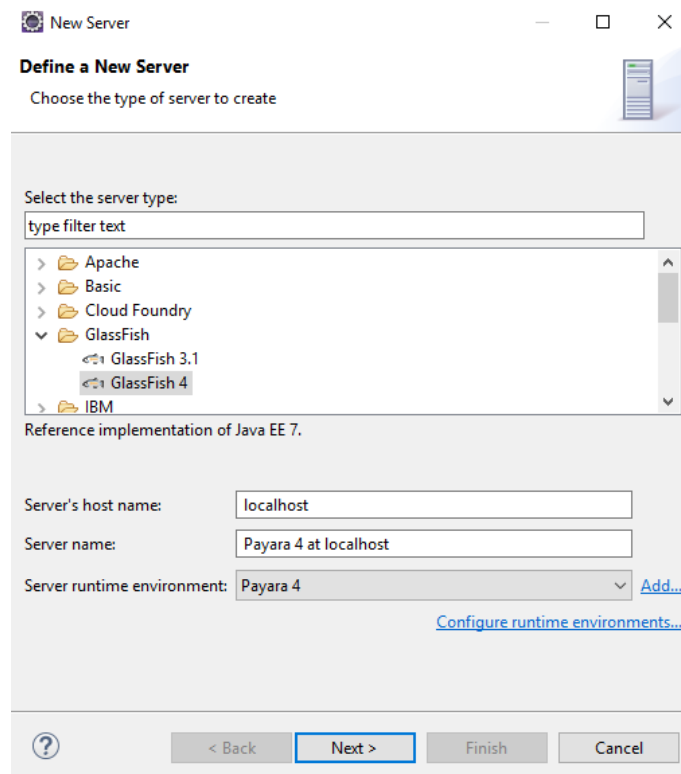


Figura 10-1- Schermata di aggiunta di un nuovo Server in Eclipse

5. Nella finestra seguente selezionare come “server type” GlassFish 4 e, se si tratta della prima esecuzione, sarà necessario aggiungere un nuovo Server runtime environment cliccando su “Add...”.
6. Nella finestra che si apre sarà necessario impostare un nome per il runtime environment, inserire il percorso per la cartella “glassfish” all’interno della directory di installazione di Payara, e specificare il percorso della propria installazione di Java.

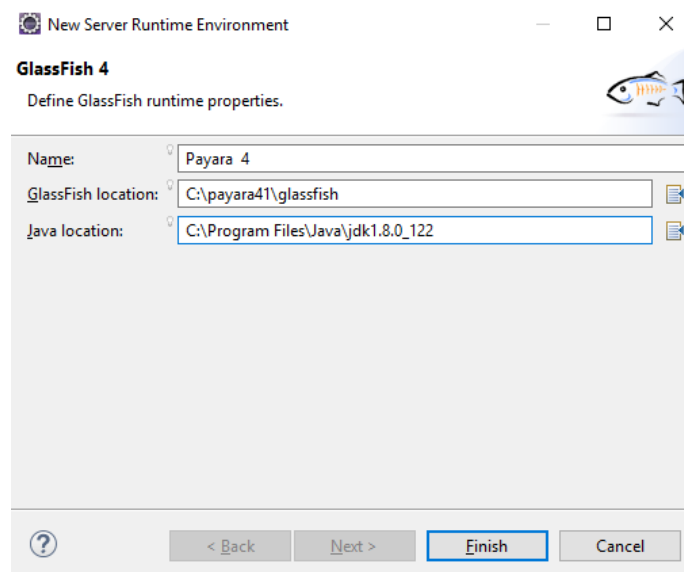


Figura 10-2- Schermata di aggiunta di un nuovo Server Runtime Environment in Eclipse

7. Una volta creato il nuovo server runtime environment, selezionarlo e proseguire nel wizard. La pagina successiva richiederà di inserire un path di dominio valido e le credenziali admin. All’interno dell’installazione di Payara è presente un dominio standard al percorso *\glassfish\domains\domain1. Per quanto riguarda le credenziali possono essere lasciate inalterate per semplificare il testing.

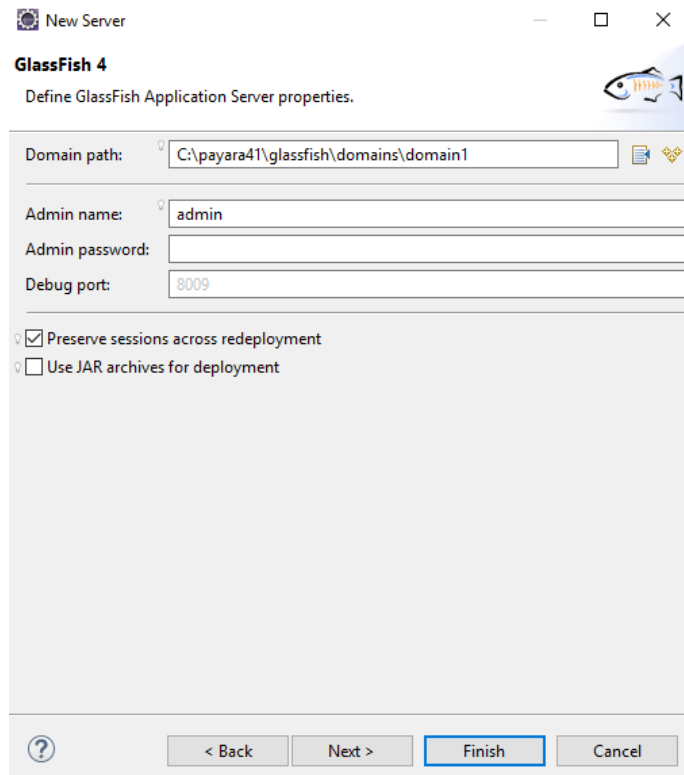


Figura 10-3- Schermata finale nel wizard di aggiunta di un nuovo Server in Eclipse

8. A questo punto la configurazione è finita e si può cliccare su Finish per completare il procedimento. Una nuova voce dovrebbe essere comparsa nella tab “Servers” di Eclipse, dalla quale sarà possibile far partire il server, fermarlo e fare deploy di applicazioni da dentro Eclipse stesso.

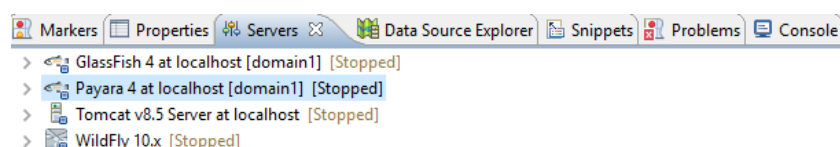


Figura 10-4- Server appena creato (Payara 4 at localhost) all'interno della tab Servers di Eclipse

Setup e deploy dell'applicazione:

1. Scaricare il codice del progetto e importarlo in Eclipse selezionando “File>Import>Existing Projects Into Workspace”.
2. Eseguire una pulizia del progetto tramite “Project>Clean...” e selezionando il progetto importato.
3. Cliccare con il tasto destro del mouse sulla voce del progetto all'interno del Project Explorer di Eclipse e selezionare “Maven>Update Project...” per

effettuare l'aggiornamento del progetto ed eventualmente scaricare le dipendenze mancanti.

4. Cliccare con il tasto destro del mouse sulla voce del progetto all'interno del Project Explorer di Eclipse e selezionare "Run As>5 Maven Build..."
5. Nella finestra che si apre inserire "clean install" nella casella di testo "Goals", quindi selezionare "Apply" e infine "Run".

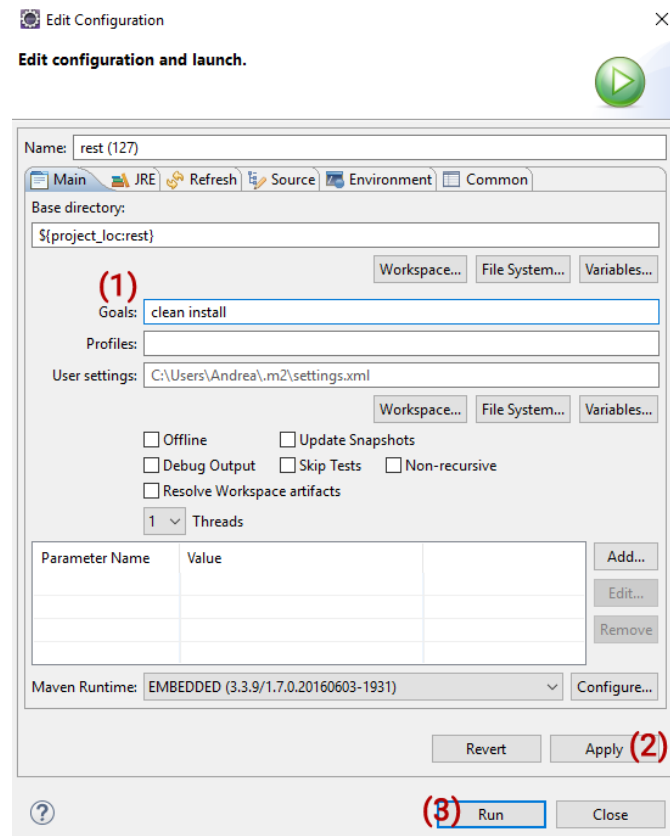


Figura 10-5- Schermata di configurazione dell'esecuzione di Maven. Per prima cosa (1) inserire "clean install" nel campo "Goals", quindi (2) premere "Apply" e infine (3) "Run"

6. Maven si occuperà di fare il build del progetto e generare il file .war dell'applicazione di cui fare il deploy.
7. Nella tab "Servers" di Eclipse cliccare con il tasto destro sul server Payara precedentemente configurato e selezionare "Add and Remove..."
8. Selezionare nella colonna sinistra l'applicazione e caricarla sul server cliccando "Add" (la voce dell'applicazione dovrebbe spostarsi nella colonna destra) e quindi "Finish"

Esecuzione dell'applicazione:

1. Avviare il database integrato di Payara tramite il comando “start-database” della console asadmin. L'eseguibile della console si trova nella cartella bin all'interno della directory di installazione (ad esempio: C:\payara41\asadmin.bat start-database). Ci si dovrà assicurare che il database sia in esecuzione ogni volta che si desidera utilizzare o testare l'applicazione.
2. Avviare il server selezionandolo nella tab “Servers” di Eclipse e premendo Ctrl+Alt+R. Il server inizierà ad avviarsi e, se si è aggiunta l'applicazione come mostrato in precedenza, verrà effettuato anche il deploy della stessa.
3. Una volta avviato il server, l'applicazione dovrebbe apparire come [Synchronized].
4. Per verificare il corretto funzionamento dell'applicazione recarsi all'indirizzo: <http://localhost:8080/rest/tuProlog/main> al quale dovrebbe comparire una descrizione XML dei servizi simile alla seguente.

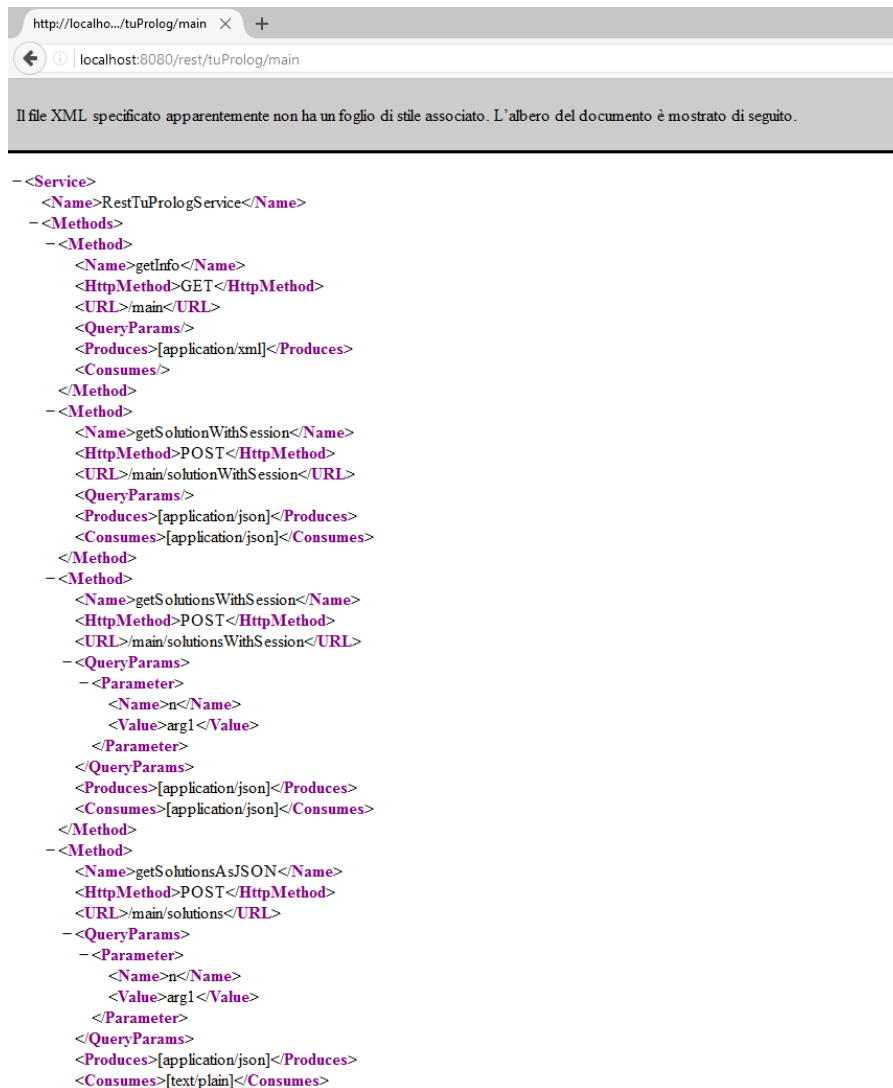


Figura 10-6- Risultati della getInfo() del servizio visualizzati su un browser, prova che l'applicazione è in esecuzione

10.3.2 Esecuzione con Maven e Payara/Payara Micro

Build del progetto con Maven:

1. Se non già presente, scaricare e installare Maven (all'indirizzo <https://maven.apache.org/download.cgi>).
2. Scaricare il codice del progetto
3. All'interno della directory principale del progetto eseguire il comando:
 “mvn clean install” per compilare e generare l'archivio “rest.war” dell'applicazione.

Deploy ed esecuzione con Payara Server Full:

1. Scaricare Payara Server Full all'indirizzo: <http://www.payara.fish/downloads>

2. Installare Payara estraendolo in una cartella (ad esempio C:\payara41)
3. Avviare il database integrato di Payara tramite il comando “start-database” della console asadmin. L’e eseguibile della console si trova nella cartella bin all’interno della directory di installazione (ad esempio: C:\payara41\asadmin.bat start-database). Ci si dovrà assicurare che il database sia in esecuzione ogni volta che si desidera utilizzare o testare l’applicazione con Payara Full.
4. Sempre utilizzando la console asadmin mettere in esecuzione il server tramite il comando start-domain (ad esempio: C:\payara41\asadmin.bat start-domain).
5. Aprire l’interfaccia grafica di amministrazione di Payara all’indirizzo <http://localhost:4848/>

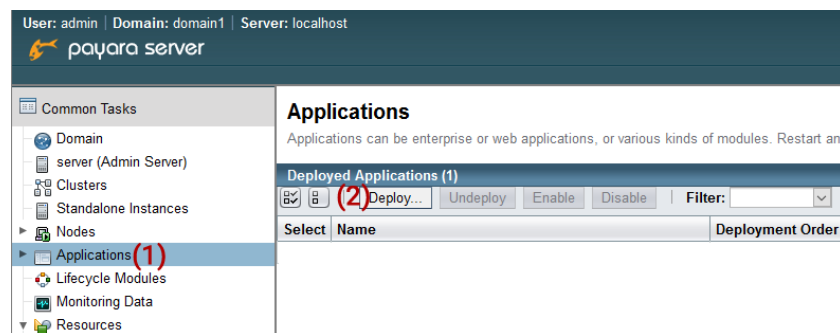


Figura 10-7- Schermata dell’interfaccia di amministrazione di Payara. Per effettuare il deploy di un’applicazione cliccare nel menù su “Applications” (1), e quindi su “Deploy...” (2) per selezionare l’archivio di cui fare il deploy

6. Selezionare “Applications” dal menù e quindi “Deploy...” per effettuare il deploy dell’archivio “rest.war”.
7. L’applicazione apparirà nella lista delle Deployed Applications e sarà raggiungibile all’indirizzo: <http://localhost:8080/rest/tuProlog/main>

Deploy ed esecuzione con Payara Micro Edition:

1. Scaricare Payara Micro all’indirizzo: <http://www.payara.fish/downloads>
2. Posizionare il file jar di Payara Micro dove si desidera e creare una cartella che sarà utilizzata come cartella di root per Payara Micro (ad esempio posizioniamo il file jar in C:\payara41 e creiamo la cartella C:\payara41\rootDir)
3. Mandare in esecuzione il server e l’applicazione con il comando:

```
java -jar payara-micro.jar --rootDir .\rootDir  
--deploy rest.war --noCluster
```

Codice 10-12- Comando per effettuare il deploy dell'applicazione e l'avvio del server Payara Micro da linea di comando

4. Fintanto che il server rimane in esecuzione l'applicazione sarà raggiungibile all'indirizzo: <http://localhost:8080/rest/tuProlog/main>

11. Bibliografia

[IOTG] Daugherty, Paul; Negm, Walid; Banerjee, Prith; Alter, Allan. “Driving Unconventional Growth through the Industrial Internet of Things”

https://www.accenture.com/mz-en/_acnmedia/Accenture/next-gen/reassembling-industry/pdf/Accenture-Driving-Unconventional-Growth-through-IIoT.pdf

[LPaaS] Calegari R.; Denti E.; Mariani S.; Omicini A. (2016) “Towards Logic Programming as a Service: Experiments in tuProlog”

<http://ceur-ws.org/Vol-1664/w14.pdf>

[IOTR] Vongsingthong, S.; Smachat, S. (2014). “Internet of Things: A review of applications & technologies”. Suranaree Journal of Science and Technology

<http://ird.sut.ac.th/e-journal/Journal/suwimonv/1403739/1403739.pdf>

[SOBJ] Kortuem, Gerd; Fahim Kawsar; Daniel Fitton; Vasughi Sundramoor (2010). “Smart Objects as Building Blocks for the Internet of Things”. IEEE Internet Computing.

<http://ieeexplore.ieee.org/document/5342399/>

[IOTS] Manyika, James, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. “The Internet of Things: Mapping the Value Beyond the Hype.” McKinsey Global Institute, June 2015.

<http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world>

[AMI1] Arribas-Ayllon, Michael. “Ambient Intelligence: an innovation narrative”

https://www.academia.edu/1080720/Ambient_Intelligence_an_innovation_narrative

[AMI2] Zelkha, Eli; Epstein, Brian; Birrell, Simon; Dodsworth, Clark (1998), “From Devices to “Ambient Intelligence””, Digital Living Room Conference.

<https://epstein.org/ambient-intelligence/>

[LBL] A. Omicini and F. Zambonelli, “MAS as complex systems: A view on the role of declarative approaches,” in Declarative Agent Languages and Technologies

http://link.springer.com/chapter/10.1007%2F978-3-540-25932-9_1

[3LA] Microsoft patterns & practices, “Three-Layered Services Application”

<https://msdn.microsoft.com/en-us/library/ff648105.aspx>

[WS] “Web Services Architecture”, W3C. 11-02-2004

<https://www.w3.org/TR/ws-arch/>

[WS-A] “Web Services Addressing (WS-Addressing)”, W3C Member Submission. 10-08-2004

<https://www.w3.org/Submission/ws-addressing/>

[REST] Fielding, Roy Thomas (2000). “Chapter 5: Representational State Transfer (REST)”. Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.

http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[JSON] Final draft Standard ECMA-262 edition 5.1, 03-2011 (Rev. 6) - Ecma 262

<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>

[MIME] Freed, N.; Borenstein, N. (November 1996). “Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies”

<https://tools.ietf.org/html/rfc2045>

[JWT] JWT homepage:

<https://jwt.io/>

[JRFC] RFC7519: JSON Web Token (JWT):

<https://tools.ietf.org/html/rfc7519>

[JWS] RFC7515: JSON Web Signature (JWS):

<https://tools.ietf.org/html/rfc7515>

[JWE] RFC7516: JSON Web Encryption (JWE):

<https://tools.ietf.org/html/rfc7516>

[PAY] Payara Application Server, “What is Payara”:

<http://www.payara.fish/about>

[GLS] Glassfish Server:

<https://glassfish.java.net/>

[PYM] Payara Micro:

http://www.payara.fish/payara_micro

[PMEM] Millidge, S. “How Micro is Payara Micro”:

<http://blog.payara.fish/how-micro-is-payara-micro>

[JRS] JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services

<https://jcp.org/en/jsr/detail?id=339>

[JSY] Project Jersey API Documentation:

<https://jersey.java.net/apidocs/latest/jersey/index.html>

[JXP] Java API for XML Processing, reference implementation homepage:

<https://jaxp.java.net/>

[JXB] Java Architecture for XML Binding, reference implementation homepage:

<https://jaxb.java.net/>

[J353] JSR 353: Java API for JSON Processing (JSON-P):

<https://jcp.org/en/jsr/detail?id=353>

[JSONP] JSON-P reference implementation homepage:

<https://jsonp.java.net/>

[GSON] Google GSON github:

<https://github.com/google/gson>

[JKSN] FasterXML Jackson github:

<https://github.com/FasterXML/jackson>

[JPER] Dreyfuss, J. (May 2015) “The Ultimate JSON Library: JSON.simple vs GSON vs Jackson vs JSONP”:

<http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/>

[JEP] JEP 198: Light-Weight JSON API:

<http://openjdk.java.net/jeps/198>

[JBIN] JSR 367: Java Api for JSON Binding (JSON-B):

<https://jcp.org/en/jsr/detail?id=367>

[JMPL] JWT libraries:

<https://jwt.io/#libraries>

[j4j] jose.4.j, bitbucket homepage:

https://bitbucket.org/b_c/jose4j/wiki/Home

[DRB] Apache Derby homepage:

<https://db.apache.org/derby/>

[JPA] JSR 317: Java Persistence 2.0:

<https://jcp.org/en/jsr/detail?id=317>

[ELK] Eclipse Link homepage:

<http://www.eclipse.org/eclipselink/>

[2PL] Home-page progetto tuProlog:

<http://tuprolog.unibo.it>

[DISI] Dipartimento Informatica – Scienza e Ingegneria, homepage:

<http://www.informatica.unibo.it/it>

[2PB] tuProlog Bitbucket homepage:

<https://bitbucket.org/tuprologteam/tuprolog>

[2PM] Maven Repository, tuProlog v3.2.0:

<https://mvnrepository.com/artifact/it.unibo.alice.tuprolog/tuprolog/3.2>

[MVN] Apache Maven homepage:

<https://maven.apache.org/>

[ECL] Eclipse IDE for Java EE Developers (v4.6 Neon)

<http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/neon2>