Università della Calabria

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di laurea in Informatica

Tesi di Laurea

Progetto ed implementazione di un approccio SaaS per BodyCloud

Relatori

Prof. Giancarlo Fortino

Dr. Giuseppe Di Fatta

Candidato

Vincenzo Pirrone

Anno Accademico 2011-2012

Indice

Abstract....1
Introduzione....2
1 Contesto....4
1.1 Workflow per l'analisi dei dati....4
1.2 Cloud Computing....6
1.2.1 Infrastructure as a Service (IaaS)....9
1.2.2 Platform as a Service (PaaS)....10
1.2.3 Software as a Service (SaaS)....11
1.3 BodyCloud....12
1.4 REST....14
1.5 Web Service....17
1.5.1 RESTful Web Service....17
2 Strumenti Utilizzati....21
2.1 Google App Engine....21
2.1.1 L'infrastruttura Cloud di Google....22
2.1.2 Runtime Environment....22
2.1.3 Datastore....24
2.1.4 Servizi....26
2.1.5 Task Queue e Cron Job....27
2.1.6 App Engine Instance e Back-end....28
2.1.7 Strumenti di sviluppo....28
2.2 Java Persistence....29
2.3 Restlet Framework....31
2.3.1 Architettura....32
2.3.2 Sviluppo di un'applicazione con Restlet....33
2.3.3 Restlet Resource....34
2.3.4 Restlet Representation....34
2.3.5 Astrazione di HTTP....35
2.3.6 Realizzazione di un web service....36
2.4 XML Data Binding....37
2.5 JxReport....41
2.6 OAuth....43

2.6.1 Attori....44
2.6.2 Access Token....45
2.6.3 Sequenza di autorizzazione....46
2.7 Maven....47
3 Progettazione....52
3.1 Problematiche....52
3.2 Scelte effettuate....54
3.3 Architettura....55
3.4 Architettura del Cloud-side....56
3.5 Progettazione del cloud-side....58
3.5.1 Casi d'uso....58
3.5.2 Domain Model....59
3.5.3 Rappresentazione dei dati....61
3.5.4 Persistence Layer API....63
3.5.5 Workflow Engine API....64
3.5.6 Web Service front-end....68
3.6 Modality....72
3.7 Sviluppo di nuovi servizi....75
3.8 Comunicazione Client-Server....77
4 Realizzazione e testing....80
4.1 Implementazione del Persistence Layer....80
4.2 Integrazione dell'Embedded Engine....82
4.3 Integrazione del Web Service....83
4.4 Autenticazione....83
4.5 Setup del progetto....85
4.5.1 weka-stripped....85
4.5.2 bodycloud-lib....86
4.5.3 bodycloud-engine....86
4.5.4 Bodycloud-server....87
4.5.5 Sviluppo di nuovi nodi....90
4.6 Caso d'uso: monitor dell'ECG....90
4.6.1 Workflow....91
4.6.2 Formato dei dati....92
4.6.3 View....92

4.6.4 Modality....93
4.6.5 Testing e analisi delle prestazioni....96
Conclusioni....98

Abstract

È stato progettato ed implementato un framework, denominato BodyCloud, per l'integrazione di Reti di Sensori indossabili con il Cloud Computing. BodyCloud permette la raccolta di dati da sensori corporali connessi via Bluetooth ad un dispositivo Android, lo streaming degli stessi dallo smartphone ad una piattaforma Cloud ospitata da Google App Engine, sulla quale è possibile definire diversi processi di elaborazione, analizzando i *data feed* di un singolo individuo o di un gruppo. L'output dei processi può essere visualizzato sugli stessi dispositivi sorgente o su altri dispositivi di monitoraggio, appositamente formattato per una visualizzazione grafica user-friendly. BodyCloud supporta l'esecuzione di applicazioni distribuite in un'ampia gamma di domini applicativi, quali health-care, sport, fitness, disaster & emergency management.

Introduzione

La crescente diffusione dei cosiddetti mobile device, vale a dire smartphone e tablet, ha portato alle persone due grandi possibilità : (1) la possibilità di avere qualunque tipo di applicazione sul palmo della mano grazie alla semplicità di sviluppo e distribuzione del software di cui i dispositivi godono, (2) la possibilità di interagire in qualunque momento con infiniti servizi online, grazie alla diffusione di reti wi-fi e 3G.

Il secondo punto è particolarmente importante perché per quante siano le applicazioni a disposizione e per quanto sia potente l'hardware del dispositivo le sue capacità rimangono limitate. L'utilizzo costante della rete permette di accedere ad una mole enorme di servizi di vario genere; basti pensare ai servizi di Google, come Maps che è in grado di elaborare un percorso in qualunque parte del mondo, Play Music che da accesso a migliaia di brani musicali o più in particolare Voice Search che è in grado di trascrivere un testo a partire dalla registrazione vocale di una frase. Sono tutti servizi che devono accedere ed elaborare un'enorme quantità di dati, operazione che sarebbe impossibile per il dispositivo stesso ma che è possibile aggirare tramite una semplice connessione alla rete.

Ma il limite delle risorse non è solo un problema per i dispositivi mobili, fino a poco tempo fa qualunque programmatore doveva fare i conti con le capacità fisiche delle macchine che eseguivano il software. Questa problematica è stata aggirata dal Cloud Computing che è stato in grado di “dissolvere” in una nuvola l'hardware dando la possibilità al software di fruire di risorse “illimitate”. Il binomio mobile device – cloud computing è ora di gran successo perché permette di avere risorse quasi illimitate a portata di mano.

Il lavoro di tesi si colloca in quest'ambito. Il BodyCloud usa un comune mobile device come gateway tra una rete di sensori e un servizio cloud, capace di analizzare i dati raccolti dai sensori e di mettere in comunicazione utenti analizzati (body) e utenti monitori (viewer). In particolare è stata progettata e realizzata la componente cloud del sistema, non focalizzandosi su un particolare insieme di servizi da offrire, ma cercando di creare un framework che permettesse a programmatori ed analisti di realizzare personalmente le loro

applicazioni BodyCloud; e che esponesse una API standard per permettere la realizzazione di applicazioni client su dispositivi mobili o desktop.

L'idea finale è di utilizzare una singola applicazione nativa su dispositivo mobile per avere accesso ad una serie di applicazioni BodyCloud (o servizi) con scopi diversi (health-care, sport, fitness, ecc.).

Il lavoro di tesi si è svolto in parte all'interno dei DEIS, Università della Calabria, e in parte a Reading (UK) presso la "University of Reading" sotto la supervisione del Dr. Giuseppe Di Fatta, nell'ambito del progetto Erasmus Placement, finanziato dalla Comunità Europea.

La tesi è suddivisa in quattro capitoli:

Capitolo 1 – Contesto

Illustra il contesto in cui il lavoro è stato svolto, vengono qui descritti i concetti di Cloud Computing, REST, web service e viene introdotto il BodyCloud.

Capitolo 2 – Strumenti Utilizzati

Vengono analizzati gli strumenti e le tecnologie con cui il lavoro è stato svolto, vale a dire Google App Engine, Restlet, XML, OAuth, Maven.

Capitolo 3 – Progettazione

Partendo dagli obiettivi da raggiungere, nel capitolo viene affrontato tutto l'iter di progettazione del software, analizzando le scelte effettuate. Vengono definiti i concetti basilari, i componenti del sistema e i diagrammi UML.

Capitolo 4 – Realizzazione e testing

Vengono descritte nel dettaglio alcune fasi di implementazione del software, e vengono illustrati i test eseguiti.

1 Contesto

1.1 Workflow per l'analisi dei dati

Estrarre informazioni utili da dati grezzi di vario tipo è un procedimento computazionale molto complesso che può richiedere l'applicazione di diversi algoritmi di natura diversa, nonché una discreta conoscenza del dominio di interesse.

Negli ultimi anni diversi software hanno portato al concepimento di tecniche visuali per facilitare l'accesso ai vasti strumenti di analisi dei dati a disposizione. Questo tipo di applicazioni, note come *pipelining tools*, permettono anche ai non esperti in programmazione di effettuare analisi complesse su grandi moli di dati, mettendo a disposizione un ambiente per creare dei *workflow*.[1]

Questi strumenti permettono di connettere consecutivamente unità di elaborazione, i nodi, creando un grafo orientato, il workflow. Ogni nodo ha una funzione ben precisa, il primo solitamente ha il compito di leggere i dati da una sorgente, come un file o un database. I dati hanno forma tabellare , ovvero ogni colonna rappresenta un attributo, e ad ogni riga corrisponde un dato, cioè una lista di valori corrispondenti agli attributi. L'input viene memorizzato internamente in strutture dati apposite e passato ai nodi successivi. I nodi intermedi tipicamente ricevono tabelle in input dai predecessori, effettuano un'elaborazione e generano una tabella in output da inviare ai successori. Le azioni dei nodi terminali invece includono scrivere il risultato finale su un file, generare un report o visualizzare un grafico. L'esecuzione del processo di analisi corrisponde all'esecuzione sequenziale dei nodi, l'esecuzione del successivo viene cioè avviata solo dopo il termine del lavoro del precedente.

Questo modello ha avuto particolare successo nel Knowledge Discovery process, che ben si presta a questo tipo di rappresentazione. Ciascuna fase del processo può infatti

essere espressa da un nodo, e software come KNIME[1] o RapidMiner[2] mettono a disposizione degli analisti una vasta gamma di strumenti di pre-processing (campionamento, filtri in base agli attributi, discretizzazione di valori numerici, raggruppamento di valori discreti, splitting e merging di tabelle) e un insieme algoritmi di data mining e machine learning (alberi di decisione, classificatori a regole, clustering).

Altro grande punto di forza di questi software è l'estensibilità. Infatti offrono sia un kit di sviluppo[2] per creare nuovi nodi, sia un repository[3][4] in cui i programmatori possono rilasciare le proprie estensioni affinché siano disponibili agli altri utenti.

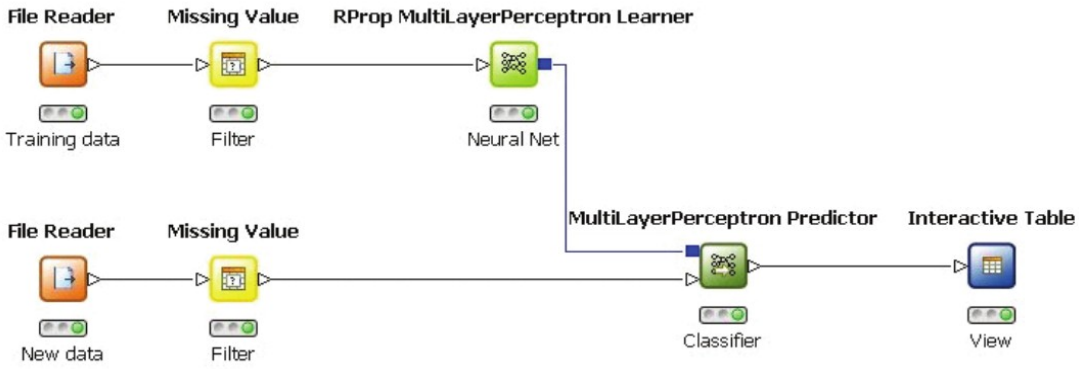

Figure 1: Simple KNIME Workflow

1 http://www.knime.org
2 http://rapid-i.com/
3 http://www.knime.org/downloads/extensions
4 http://rapid-i.com/content/view/202/206/lang,en/

1.2 Cloud Computing

Inizialmente il termine cloud veniva utilizzato in riferimento alle reti di telecomunicazione e alla rete internet in quanto queste erano rappresentate come nuvole negli schemi tecnici, a indicare aree in cui le informazioni venivano spostate ed elaborate; tuttavia, ciò avveniva senza che l'utente sapesse esattamente quel che stava realmente accadendo. Questa è la caratteristica principale del cloud computing: il cliente richiede e riceve informazioni o altre risorse senza sapere dove risiedono o secondo quale meccanismo il servizio nel cloud soddisfa la richiesta.

Una definizione formale e ampiamente adottata è stata elaborata dal National Institute of Standards and Technology:

> *Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.* [3]

Ovvero, un modello per abilitare un accesso conveniente e su richiesta a un insieme condiviso di risorse computazionali configurabili (ad esempio reti, server, memoria di massa, applicazioni e servizi), che possono essere rapidamente procurate e rilasciate con un minimo sforzo di gestione o di interazione con il fornitore del servizio.

Di per se tale definizione non dice nulla, non specifica un'architettura né uno standard ma solo un insieme di caratteristiche. Il termine è stato utilizzato per la prima volta dal CEO di Google, Eric Schmidt il 9 Agosto 2006 in una conferenza:

> *it starts with the premise that the data services and architecture should be on servers. We call it cloud computing – they should be in a 'cloud' somewhere. And that if you have the right kind of browser or the right kind of access, it doesn't matter whether you have a PC or a Mac or a mobile phone or a BlackBerry or what have you – or new devices still to be developed – you can get access to the cloud…[4]*

Solo poche settimane dopo il discorso di Schmidt (il 25 Agosto) Amazon lancia la beta di EC2[5] (Elastic Compute Cloud), un vero punto di riferimento nel cloud computing, anche se il primo servizio che può godere del termine risale al lontano 1999, offerto da Salesforce.com. Ma il cloud non rappresenta una novità tecnologica, può essere visto

5 http://aws.typepad.com/aws/2006/08/amazon_ec2_beta.html

invece come l'utilizzo congiunto di diverse tecnologie e concetti uniti sotto un unico nome, il quale, per la sua semplicità e originalità, è facilmente diventato una buzzword.[5]

Il concetto basilare nasce dal problema di un ente di dotarsi di un'infrastruttura IT per determinati scopi. A tal bisogno è necessario far fronte a costi notevoli e solitamente le risorse di calcolo rimangono in gran parte inutilizzate. Da qui l'idea di *Utility Computing*, ovvero l'utilizzo di risorse di calcolo altrui (tipicamente tempo-macchina) su richiesta e con una forma di pagamento a consumo. Il termine sta proprio ad indicare un'analogia tra le risorse computazionali e i servici pubblici come acqua, luce e gas, le *public utilities* appunto, rimarcando un'affermazione di John McCarthy del 1961: "*computation may someday be organized as a public utility"*[6]. In questo modo chi si è dotato di un'infrastruttura può sfruttarla ulteriormente ed ammortizzare i costi più velocemente, divenendo un fornitore di servizi; il consumatore invece può ridurre notevolmente i costi derivanti dall'acquisizione dell'hardware, pagando invece esclusivamente per le risorse effettivamente utilizzate.

Il successo di questo modello è dovuto soprattutto all'evoluzione della virtualizzazione dell'hardware, ovvero la tecnologia che permette di simulare tramite software l'esecuzione di una macchina virtuale, su cui eseguire a sua volta software arbitrario. Ciò permette al fornitore del servizio di offrire una risorsa ben definita e limitata (una o più macchine virtuali appunto), semplice da monitorare ed isolata, in modo che l'esecuzione del software sulla macchina virtuale non intacchi in nessun modo la sicurezza e le prestazioni della macchina ospitante, problemi che invece si presenterebbero con un approccio multi-utenza. Il consumatore d'altro canto ha la possibilità di usufruire di un'infrastruttura virtuale su cui ha il pieno controllo (SO, rete, ecc.)[7].

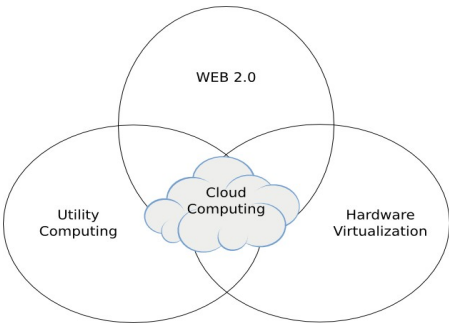

Figure 2: Convergence of various advances leading to the advent of cloudcomputing

Il cloud computing vero e proprio nasce dall'incontro di queste idee col Web 2.0. La risorsa cloud non è mero tempo-macchina concesso al consumatore, ma comprende lo strato software sovrastante personalizzato a seconda delle necessità del consumatore; essa può usufruire da spazio di archiviazione, server, applicazione o altro. Inoltre il servizio è accessibile tramite internet attraverso i meccanismi standard del web (HTTP, REST, SOAP, JSON, XML) che promuovono lo sviluppo e l'utilizzo di client differenti, nonché l'accesso tramite diversi tipi di dispositivi (laptop, palmari, smartphone), in modo completamente self-service. Tutte le risorse nel cloud sono organizzate e gestite come un pool comune condiviso, con risorse differenti assegnate e riassegnate dinamicamente agli utenti che ne fanno richiesta. La “quantità” delle risorse utilizzate può essere modificata rapidamente e in ogni momento in base alle necessità dell'utente o del sistema in modo completamente automatico. È proprio l'insieme di questi fattori che rende tra l'altro difficile stabilire dove i dati risiedono e dove avviene l'elaborazione, si può quindi immaginare che tutto avvenga in una nuvola, in un processo di cui il consumatore non deve avere la benché minima preoccupazione.

Ma cosa comporta tutto ciò? Quali sono i possibili utilizzi? E i vantaggi? Innanzi tutto scompare quella fase in cui l'utente acquisisce i propri mezzi, in un certo senso l'informatica del Cloud è l'informatica senza computer, tutto sta nella nuvola. Ora l'utente non ha più bisogno di utilizzare il proprio PC, può usufruire delle sue applicazioni da qualunque postazione. La piccola azienda che ha bisogno di una piattaforma on-line non deve porsi più il problema di investire sui propri server o cercare un servizio con le caratteristiche adeguate, generalmente una soluzione Cloud fornisce tutto il necessario ad un costo ridotto; non solo, mentre spesso i servizi standard risultano inadeguati, le risorse insufficienti o i costi di gestione troppo alti, la promessa del Cloud è di avere sempre e comunque il tipo di risorsa adatto, con tempi di cambiamento strettissimi. Ciò che prima richiedeva settimane (acquisire nuove macchine, acquisire personale, configurare il software) ora si è ridotto a pochi minuti e quelli che erano i costi più vari (hardware, tecnici, elettricità, raffreddamento, consulenze, outsourcing) ora sono tradotti in una sola formula, “si paga quello che si usa”. Dal punto di vista del provider i benefici non sono minori, dotarsi di un'infrastruttura Cloud vuol dire riuscire a soddisfare varie tipologie di esigenze. Con gli stessi server a disposizione il modello Cloud permette di tenere pronti diversi tipi di ambiente ed essere sempre preparati a soddisfare le richieste dei clienti.

Sempre secondo la definizione del NIST i servizi cloud possono essere divisi in tre

classi a seconda del livello di astrazione offerto, essi sono: Infrastructure as a Service, Platform as a Service, Software as a Service[3] . I livelli di astrazione possono anche essere visti come un'architettura a strati, il livello inferiore consiste in un pool di macchine virtuali su cui poggia una piattaforma cloud capace di ridimensionare l'infrastruttura e distribuire il carico dell'applicazione finale[5].

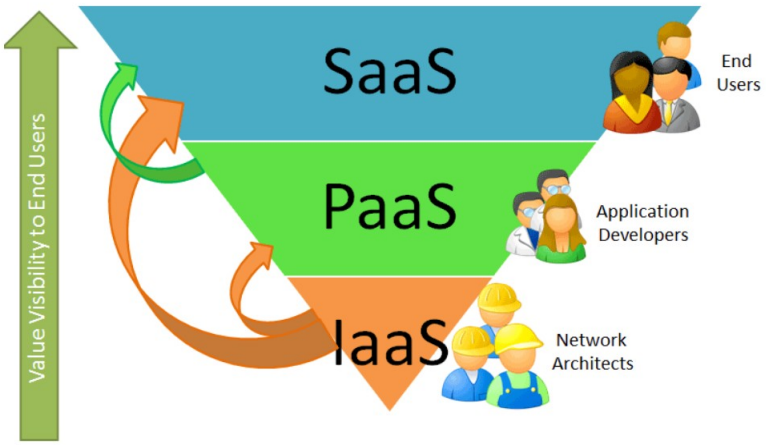

Figure 3: Cloud Computing Service Models

1.2.1 Infrastructure as a Service (IaaS)

Il servizio offerto all'utente consiste nella possibilità di usufruire di capacità di calcolo, network, storage e altre risorse informatiche di basso livello su richiesta. Tipicamente il provider mette a disposizione una API per il monitoring e il controllo di un'infrastruttura virtuale, in pratica permette il controllo di hardware (seppur virtuale) tramite software.

Amazon si è imposto come punto di riferimento nel settore con EC2, un web service che permette di richiedere server virtuali con diversa potenza di calcolo e software stack personalizzato. Il consumatore può scegliere tra diverse immagini (AMI – Amazon Machine Image) di VM (virtual machine), ciascuna preconfigurata per un determinato scopo, e utilizzarle per creare delle istanze. Ciascuna istanza ha una dimensione che può essere selezionata tra quelle disponibili (small, medium, large, xlarge) che ne determina la potenza di calcolo e il costo orario. EC2 fornisce anche avanzati meccanismi di configurazione delle istanze (indirizzi IP, firewall, DNS, load balancing), il tutto tramite web-api o interfaccia web.

1.2.2 Platform as a Service (PaaS)

Affinché un software possa godere dell'elasticità un'infrastruttura cloud è necessario un middleware capace di controllarla. Lo scopo del PaaS è quello di fornire un ambiente di esecuzione di applicazioni, in grado di distribuire il carico di lavoro tra le macchine virtuali dell'infrastruttura sottostante, ottimizzando l'utilizzo di quest'ultima. Questa tipologia è rivolta a programmatori che, in questo modo, non devono curarsi dell'hardware su cui verrà eseguita la loro applicazione; esso sarà rappresentato da una nuvola e l'applicazione sarà sempre in grado di soddisfare il suo carico di lavoro.

Il più popolare PaaS è Google App Engine, realizzato dal provider americano e pensato soprattutto per l'esecuzione di applicazioni web. Essendo un prodotto proprietario le sue specifiche non sono pubbliche, ma un ambiente simile è stato implementato dal software open-source AppScale, che fornisce un buon esempio di PaaS.

AppScale è un runtime system scalabile e distribuito, sviluppato alla University of California, pensato per essere eseguito su un cluster cloud su Amazon EC2 o Eucalyptus (implementazione open-source delle API di EC2). Un'applicazione distribuita su AppScale viene replicata su tutti i nodi del cluster, il nodo master funge anche da load balancer, smistando le richieste alle varie VM e, all'occorrenza, è in grado di ordinare nuovi nodi attraverso chiamate al cloud provider. Dal momento che non è possibile stabilire quale VM soddisfa ciascuna richiesta AppScale non permette alle guest application di aprire socket, ma consente solo richieste HTTP in quanto stateless. Sempre per lo stesso motivo non è permesso l'accesso al filesystem, né l'utilizzo di un database centralizzato in quanto non scalabile. AppScale invece supporta l'utilizzo di database distribuiti come HBase, Hypertable, Apache Cassandra, Voldemort, MySQL Cluster e MongoDB[8].

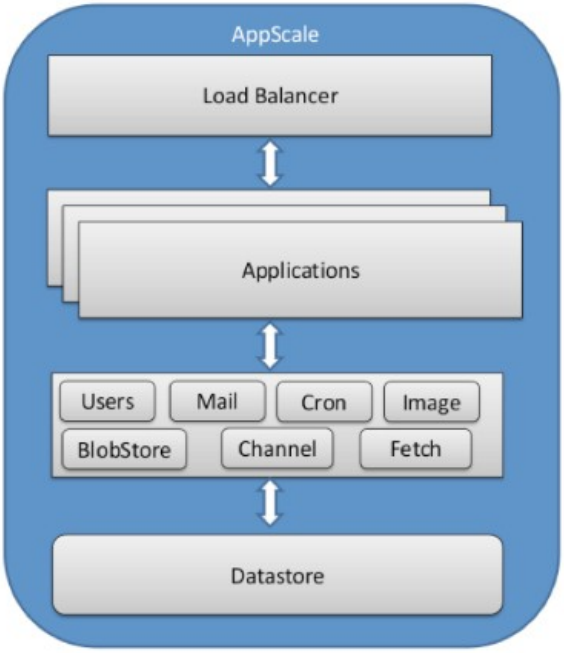

Figure 4: AppScale architecture

1.2.3 Software as a Service (SaaS)

Il servizio offerto all'utente consiste nella capacità di utilizzare applicazioni installate non sul proprio PC ma al di sopra di uno stack cloud. L'accesso all'applicazione viene effettuato tramite programmi client, spesso tramite browser web, cosa che consente agli utenti di spostare la loro produttività da applicazioni locali a software online che mantengono le stesse funzionalità. Il SaaS è il cloud destinato all'utente finale, un prodotto completo pronto per essere utilizzato.

SaaS è diventato un metodo di distribuzione di diversi software di produttività, in quanto permette alle aziende di richiedere al cloud provider una versione personalizzata di un'applicazione raggiungibile attraverso un dominio specifico, senza la necessità di installazioni multiple, con aggiornamento centralizzato, ma soprattutto con particolari funzionalità volte a facilitare il lavoro di gruppo. Questo modello di business è adottato da Google Apps e Salesforce.com.

Altri ambiti di utilizzo del SaaS sono successivamente stati offerti dalla proliferazione

dei dispositivi mobile, con la nascita di numerosi servizi di sincronizzazione e storage dei dati come Dropbox e iCloud.

Tale modello è stato anche di ispirazione alla nascita di ChromeOS, un sistema operativo concepito per eseguire esclusivamente applicazioni web.

1.3 BodyCloud

Una rete di sensori consiste in un insieme di nodi sensori, spazialmente distribuiti, che cooperano al monitoraggio di condizioni fisiche, ambientali o umane, come temperatura, pressione atmosferica, movimento, battito cardiaco e pressione sanguigna. I nodi di una rete di sensori producono una grande quantità di dati, inerenti al contesto monitorato. La raccolta e l'analisi di dati di questo tipo può portare al miglioramento di diversi servizi quali domotica, healthcare analysis, previsioni del tempo e controllo del traffico.

Specifiche reti di sensori adibite al monitoring di condizioni umane sono definite Body Sensor Networks (BSN), e hanno diverse aree di applicazione che spaziano dalla medicina alla misurazione di prestazioni sportive, fino al gaming e al social networking. Il loro utilizzo è reso ancor più semplice con la diffusione di dispositivi mobile con software sempre più versatile (es. Android, iOS, Windows Phone) e console che integrano sensori nei dispositivi di input (Nintendo Wiimote, PlayStation Move, Microsoft Kinect). In un comune ambito applicativo, gli assistiti sono monitorati da una BSN capace di raccogliere dati non solo per essere processati in tempo reale, ma anche per essere memorizzati in un repository remoto per essere analizzati da esperti del contesto.

La grande quantità dei dati raccolti dai sensori, unita alla limitata capacità di calcolo dei dispositivi di monitoring e alla necessità di uno storage remoto centralizzato, pongono diversi problemi di realizzazione. Tuttavia il Cloud Computing ha portato alla possibilità di realizzare un sistema software senza la necessità di occuparsi dell'infrastruttura hardware. L'architettura risultante dall'integrazione tra BSN e Cloud Computing è stata definita *BodyCloud*. [9]

Gli aspetti fondamentali dell'architettura *BodyCloud* sono i seguenti:

Gestione dei dati

Le attività da svolgere sui dati possono riguardare diverse sorgenti, distanti tra di loro, e diverse rilevazioni avvenute in momenti diversi. Pertanto i dati devono essere raccolti, memorizzati e convogliati per il processo in modo efficiente.

Processing

In alcuni casi è necessario processare in tempo reale dati eterogenei provenienti da diverse fonti, questo tipo di analisi richiede grandi capacità di calcolo.

Invocazione dei servizi

I dati processati di una BSN vengono associati a meta-dati riguardo alla procedura di derivazione (derivation), all'ente e alla motivazione dell'acquisizione (agency), e a come possono essere distribuiti (rights). Il processo viene eseguito tramite la formazione automatica di workflow e l'invocazione di servizi.

Analisi

i dati collezionati dalle BSN possono essere importati all'interno di tool di analisi e modellazione per essere utilizzati nell'esecuzione di applicazioni e sistemi decisionali. Le operazioni di analisi dipendono da tecnologie di storage e middleware capaci di effettuare calcoli in modo estremamente rapido.

Applicazioni pratiche di BSN puntano a migliorare la qualità della vita fornendo assistenza medica continuativa, in tempo reale, non invasiva ed economica. Applicazioni in cui le BSN possono rivelarsi estremamente utili includono la preventiva diagnosi o la prevenzione di malori e malattie (es. attacchi di cuore, Parkinson, diabete, asma); riabilitazioni post-operatorie; rilevamento dei movimenti in giochi interattivi; riconoscimento cognitivo ad emotivo per assistenza alla guida o interazioni sociali; ed assistenza medica durante le emergenze (es. terremoti, attacchi terroristici).

1.4 REST

REST (REpresentational State Transfer) può essere definito come un insieme di linee guida per l'architettura di sistemi distribuiti consoni agli standard del World Wide Web, definito da Roy Fielding (uno dei principali autori della specifica HTTP) nella sua tesi di dottorato nel 2000.[10]

Fielding parte da un insieme di constraint che il sistema deve avere per poi definirne gli elementi architetturali:

Client-Server

Il paradigma più usato nelle architetture delle applicazioni network-based. Un componente (un nodo del sistema) detto server offre un insieme di servizi. Un secondo componente detto client invia una richiesta al server in base al servizio di cui necessita. La base di questo constraint è rappresentata dalla separazione delle funzionalità, in modo da semplificare la realizzazione e la scalabilità del server, e il riutilizzo del codice lato client. Ad esempio lo storage dei dati non deve essere demandata al client così come il server non deve occuparsi dell'interfaccia utente. Da notare che la distinzione tra client e server è puramente logica e relativa, un componente può ad esempio essere un client per un determinato server a contemporaneamente offrire servizi per un altro tipo di client.

Stateless

Comunicazione senza stato, ogni singola richiesta del client deve quindi contenere tutte le informazioni necessarie al server per essere elaborata. La principale ragione di questa scelta è la scalabilità: mantenere lo stato di una sessione ha un costo in termini di risorse sul server e all’aumentare del numero di client tale costo può diventare insostenibile. Inoltre, con una comunicazione senza stato è possibile creare cluster di server che possono rispondere ai client senza vincoli sulla sessione corrente, ottimizzando le prestazioni globali dell’applicazione.

Cache

Il client può essere in grado di memorizzare le risposte del server in modo da evitare di ripetere le medesime richieste. Pertanto ogni risposta che rimane costante deve essere marcata come cacheable in modo da permette al client di utilizzare tale funzionalità.

Sistema stratificato

Il sistema può essere rappresentato da diversi livelli di astrazione, in cui ogni livello offre funzioni al livello superiore e utilizza quelli del livello inferiore. Tale concetto architetturale, trasposto alla realizzazione di sistemi distribuiti, implica che la presenza di componenti a livello più basso non deve influire sul funzionamento dei componenti di livello più alto. Ad esempio la comunicazione tra client e server, essendo puramente applicativa, non deve subire alcun cambiamento se fra essi vi si frappone un proxy o un gateway, componenti che regolano il traffico di rete.

Code-On-Demand

Le funzionalità del client possono essere estese scaricando opportuno codice eseguibile dal server.

Interfaccia uniforme

I componenti devono mantenere un'interfaccia uniforme basata su identificazione e rappresentazione delle risorse. Ciò implica che il client non deve avere nessuna conoscenza dei dettagli implementativi del server, né deve essere necessario utilizzare tipi di dato non-standard definiti da quest'ultimo.

Quest'ultimo constraint, ovvero la REST API, è la base per definire gli elementi architetturali di REST. La comunicazione consiste in uno scambio di informazioni. Per REST ogni informazione a cui può essere assegnato un nome, ogni concetto con un significato all'interno del dominio applicativo viene definito **risorsa**. Esempi di risorsa possono essere documenti, immagini, ma anche entità dinamiche (es. la temperatura di oggi a Roma). Tale astrazione permette di raggruppare diverse sorgenti di informazione senza però definire un particolare tipo o implementazione. Nell'interazione tra componenti, tali risorse devono innanzi tutto essere identificate, pertanto REST utilizza un **identificatore** univoco per selezionare una particolare risorsa involta nella comunicazione.

I componenti del sistema effettuano operazioni sulle risorse scambiandosi una **rappresentazione** di quest'ultima. Una rappresentazione consiste in un insieme di byte (i dati) e un insieme di meta-dati necessari per descrivere i dati. Le operazioni sulle risorse vengono descritte da dati di controllo, che specificano quindi l'azione richiesta o il significato della risposta. Dai dati di controllo dipende anche il significato della rappresentazione, questa può ad esempio descrivere lo stato corrente della risorsa identificata sul server, lo stato desiderato dal client, il valore di un'altra risorsa necessaria all'elaborazione, o un messaggio di un errore. Il formato della rappresentazione è strettamente correlata al suo scopo, alcuni formati sono più adatti per la visualizzazione e presentazione (es. HTML), altri per l'elaborazione (es. XML). Una risorsa può anche avere diverse rappresentazioni, e lasciare la scelta al client su quale utilizzare.

Altro aspetto di questo tipo di interfaccia è che solitamente le risorse sono correlate tra loro, per REST tale correlazione deve essere espressa direttamente nelle rappresentazioni utilizzando link ipertestuali. Una REST API deve essere quindi *hypertext-driven*, principio che pone l'accento sulle modalità di gestione dello stato dell'applicazione. In sostanza, tutto quello che un client deve sapere su una risorsa e sulle risorse ad essa correlate deve essere contenuto nella sua rappresentazione o deve essere accessibile tramite collegamenti ipertestuali. Grazie a questo principio un client REST può anche avere una conoscenza limitata delle risorse del server e "scoprirne" altre utilizzando i link, ciò permette una forte separazione tra client e server e concede a quest'ultimo la capacità di evolvere indipendentemente.

Una singola richiesta al server è formata quindi come segue:

1. l'id univoco della risorsa
2. dati di controllo che definiscono lo scopo della richiesta ed eventuali parametri
3. eventualmente la rappresentazione di una risorsa

Invece per quanto riguarda la risposta:

1. dati di controllo che specificano l'esito della richiesta o attributi della risposta (ad esempio se la risposta è cacheable)
2. eventualmente la rappresentazione dello stato attuale della risorsa indicata dal client

1.5 Web Service

Il W3C definisce un web service come un sistema software progettato per supportare l'interoperabilità tra diversi elaboratori su di una medesima rete, utilizzando come protocollo di comunicazione prevalentemente HTTP.

L'idea nasce dal concetto stesso di World Wide Web inteso come una rete di informazioni legate tra loro. Infatti ogni sito web (in particolare ogni applicazione web) offre di per se un servizio ai suoi utenti, tramite il browser che non è altro che il tramite (in quanto conosce il protocollo HTTP) e l'interprete (in quanto capace di renderizzare HTML ed eseguire JavaScript). Il passo successivo compiuto dai web service è quello di sostituire l'HTML come formato di codifica dei contenuti, con altri formati (in particolare XML) più adatti a rappresentare informazioni e più facilmente interpretabili dalle applicazioni. Ciò che un web service offre in più rispetto ad una normale applicazione web è un'interfaccia standard che permette la facile realizzazione di software atto ad usufruire delle funzioni offerti dal servizio. Così come le web application costituiscono il web destinato agli umani, così i web service costituisco il web programmabile destinato ai software.[11]

I web service si sono diffusi rapidamente perché è possibile realizzarli facilmente partendo da un'applicazione web già esistente; e poggiano su un protocollo general-purpose, semplice, scalabile e che rende possibile l'interoperabilità tra software di diverse piattaforme.

1.5.1 RESTful Web Service

Un RESTful web service è un web service implementato secondo i principi di REST. Applicare REST a questo campo risulta particolarmente semplice ed efficace perché HTTP presenta tutti i principi guida di un'interfaccia REST. Infatti HTTP è un protocollo basato su un meccanismo di richiesta-risposta stateless in una comunicazione client-server, e sul trasferimento di rappresentazioni di risorse remote identificate da URI.

Oltre ai vincoli di REST un web service di questo tipo deve sottostare ad altri tre principi:[12]

- Utilizzo esplicito dei metodi HTTP
- Utilizzo di URI strutturati come una gerarchia di cartelle
- Utilizzo di XML o JSON per le rappresentazioni

Utilizzo esplicito dei metodi HTTP

Nell'interazione con un RESTful web service lo scopo della richiesta è espresso esplicitamente dai metodi HTTP, per tanto il server deve rispettare la semantica del metodo utilizzato, mappando ad ogni metodo un'azione ben precisa:

`GET`
Richiede la rappresentazione della risorsa indicata nell'URI.

`PUT`
Invia la rappresentazione desiderata della risorsa indicata nell'URI.

`POST`
Richiede che il server accetti la rappresentazione contenuta nella richiesta come nuova risorsa subordinata alla risorsa indicata nell'URI.

`DELETE`
Elimina la risorsa indicata nell'URI.

Un RESTful web service consiste in pratica in una web API in cui un URI e un metodo HTTP identificano una singola azione da svolgere, e le rappresentazioni codificano i parametri su cui operare e il risultato dell'operazione. Gli URI sono composti dall'URL del web service, detto anche endpoint, e il path di una risorsa.

Utilizzo di URI strutturati come una gerarchia di cartelle

Gli URI identificano le risorse, pertanto devono riflettere la loro tipologia e gerarchia. Ad esempio un servizio di e-commerce che vende prodotti di diverse categorie potrebbe strutturare l'URI nel seguente modo:

```
http://www.myservice.org/{category}/{product}
```

In questo modo è l'URI stesso ad indicare l'esistenza di una tipologia di risorsa, la categoria, che a sua volta contiene un altro tipo di risorsa, il prodotto. Se la categoria è identificabile con una stringa e il prodotto con un identificatore numerico l'URI di una richiesta sarà simile al seguente:

```
http://www.myservice.org/books/123
```

In generale l'URI di un web service dovrebbe essere composto da nomi o numeri, non dovrebbe avere alcuna estensione (.jsp, .php, .asp) e non contenere alcuna query string.

Utilizzo di XML o JSON per le rappresentazioni

Per quanto riguarda le rappresentazioni, un web-service dovrebbe utilizzare esclusivamente XML per la sua diffusione, semplicità, espandibilità e portabilità. Un documento XML è infatti facilmente interpretabile su qualsiasi piattaforma, ma allo stesso tempo è *human readable*, in più i tag offrono una certa semantica dei dati rappresentati, perciò è facile mappare una qualsiasi struttura dati o il record di un database in questo formato.

Una rappresentazione deve inoltre contenere collegamenti ipertestuali alle risorse correlate, ovvero URI che le identificano, ed XML si presta bene anche per questo scopo.

```
<?xml version=”1.0”?>
<category name=”books”>
     <product name=”A” href=”/books/1”/>
     <product name=”B” href=”/books/2”/>
     <product name=”C” href=”/books/3”/>
</category>
```

Snippet 1: REST XML Representation Example

Nell'esempio mostrato nello snippet 1 una categoria può essere rappresentata come una lista dei prodotti che contiene (ovvero una lista delle risorse di tipo prodotto collegate alla risorsa stessa):

Più in generale però un web service può supportare qualsiasi tipo di dato identificabile con un internet media type[6], ad esempio JSON è un altro popolare formato che offre le stesse caratteristiche di XML; lasciando al client la possibilità di scegliere il formato preferito. A tal proposito è possibile utilizzare l'header HTTP `Content-Type` per specificare il formato trasportato, e il client può utilizzare l'header `Accept` per specificare i media type supportati. Client e server possono quindi mettersi d'accordo sul formato da utilizzare, meccanismo noto come content negotiation, cosa che riduce i vincoli implementativi del client.

6 http://www.iana.org/assignments/media-types

Autenticazione

Discorso a parte merita l'autenticazione. Tradizionalmente le applicazioni web implementano un proprio meccanismo di autenticazione e utilizzano i cookie per rappresentare la sessione. Questo approccio naturalmente non è RESTful, perché la comunicazione non è stateless.

Anche per questo problema i web service si affidano a funzionalità di HTTP, vale a dire *HTTP Basic Authentication* o *HTTP Digest Authentication*. Per quanto riguarda la Basic Authentication il client invia username e password codificati in base64 nell'header authorization.

Naturalmente l'invio delle credenziali con la semplice codifica Base64 non è sufficiente garanzia di riservatezza. L'uso dell'*HTTP Digest Authentication* è sicuramente un passo avanti per evitare la trasmissione in chiaro della password, ma non è esente da possibili attacchi.

La soluzione che in genere andrebbe adottata consiste nella creazione di un canale di trasmissione sicuro, come può essere *HTTPS*, cioè *HTTP over SSL/TLS*. L'adozione di HTTPS garantisce la riservatezza e l'integrità nella trasmissione delle informazioni, coprendo gli altri aspetti di sicurezza.[13]

2 Strumenti Utilizzati

2.1 Google App Engine

Google App Engine (GAE) è un Platform As A Service (PaaS) che permette l'esecuzione di applicazioni web sull'infrastruttura Cloud di Google. GAE presenta le due principali caratteristiche di un PaaS:

Scalabilità

All'aumentare delle richieste ricevute da un'applicazione web ospitata, nuove risorse vengono allocate automaticamente e messe a disposizione della stessa. Il meccanismo è trasparente rispetto all'applicazione, la quale non deve curarsi delle risorse utilizzate.

Pay-per-Use

Si pagano solo le risorse utilizzate. Le risorse a pagamento includono banda consumata (in gigabyte), uptime orario delle istanze, storage occupato, operazioni effettuate sul datastore. Un'applicazione può anche essere eseguita gratuitamente, purché non superi determinati limiti. Altrimenti è possibile stabilire un budget giornaliero che verrà speso, se necessario, per soddisfare la scalabilità dell'applicazione.

2.1.1 L'infrastruttura Cloud di Google

Affinché applicazioni del calibro di Google Search e Google Mail possano essere disponibili in tutto il mondo ed ad un vasto bacino di utenza Google mantiene diversi data center, ognuno con centinaia di computer, sparsi per il globo, collegati tra loro da connessioni di rete velocissime. Ogni aspetto dei data center è curato nei dettagli: costi, prestazioni, affidabilità, dispendio energetico, ecc. e la struttura interna di un data center può cambiare anche molto velocemente.

Il risultato è un'infrastruttura così complessa e dinamica, che è impossibile per i programmatori tener traccia di tutte le configurazioni e tutti i cambiamenti. Sarebbe complicatissimo se Gmail fosse programmato per essere eseguito su un particolare server o dovesse essere installato all'occorrenza su una macchina. Ogni server può essere spento o riavviato in qualunque momento, persino un intero data center può essere disattivato per manutenzione o per un qualunque altro motivo.

Quindi Google ha sviluppato un apposito framework, un livello di astrazione, che nasconde tutti i dettagli riguardo a dove si trovano i dati o a quale software è in esecuzione su quale server di quale data center. Quello che si ottiene è un ambiente in cui i programmatori possono sviluppare il software senza preoccuparsi di dove esso verrà eseguito, e le risorse sottostanti possono essere dinamicamente reallocate in base ai cambiamenti di necessità.

Verosimilmente GAE parte da quest'idea per offrire le stesse potenzialità (o quasi) al pubblico, creando un servizio su misura utilizzabile da chiunque.[14]

2.1.2 Runtime Environment

Un'applicazione rilasciata su GAE risponde a richieste HTTP. Quando GAE riceve una richiesta, identifica l'applicazione dall'indirizzo web, che può essere un sotto dominio di *.appspot.com* (dominio fornito gratuitamente ad ogni applicazione) o un qualsiasi altro dominio aggiuntivo registrato dall'utente. La piattaforma seleziona poi un server tra i tanti disponibili per gestire la richiesta, il server selezionato è quello che probabilmente riesce a elaborare la risposta nel più breve tempo possibile. Su quel server viene quindi creata un'istanza dell'applicazione pronta a gestire la richiesta.

Dal punto di vista dell'applicazione, la sua esistenza inizia quando riceve una riceve una

richiesta, e termina subito dopo aver restituito una risposta. L'applicazione non è in grado di mantenere internamente alcuno stato tra le richieste, anche se GAE fornisce diversi metodi per salvarlo esternamente. Senza la necessità di mantenere lo stato di un'applicazione, GAE può distribuire il traffico tra quanti più server ritiene necessario, in modo da mantenere costante il carico di un'applicazione a prescindere dalle richieste che deve gestire.[15]

In realtà GAE cerca di ottimizzare la gestione delle istanze, perciò una volta creata un'istanza verosimilmente non verrà subito distrutta ma avrà la possibilità di gestire altre richieste, anche contemporaneamente, onde evitare inutili e dispendiose inizializzazioni. Nuove istanze vengono create quando necessario, e istanze inutilizzate vengono terminate. Quando una richiesta deve essere elaborata, Google seleziona quale server o data center ha un'istanza dell'applicazione in esecuzione, o verosimilmente crea una nuova istanza in un data center geograficamente vicino al client che non sia in quel momento sovraccarico. Il punto cruciale è che una singola applicazione non ha idea di dove sia fisicamente in esecuzione o di quante copie siano attive in quel momento.[14]

Altro aspetto importante da considerare è che le applicazioni vengono eseguite in una *sandbox*. La sandbox limita l'accesso alle funzionalità del sistema operativo sottostante: un'applicazione può leggere i propri file, ma non può scrivere sul filesystem né accedere a file di altre applicazioni; non può creare nuovi processi e nemmeno aprire connessioni di rete che non siano HTTP. Ciò permette ad GAE di eseguire diverse applicazioni su uno stesso server senza che una possa compromettere un'altra. Inoltre limitando l'accesso al sistema operativo, viene anche limitato l'ammontare di cicli di CPU e memoria che una singola richiesta può utilizzare. Infine, affinché l'allocazione delle risorse sia efficiente, un'istanza non può occupare risorse per un tempo indefinito, ogni richiesta deve essere elaborata entro 60 secondi, altrimenti il processo viene terminato.

L'accesso al sistema operativo è limitato anche perché il runtime environment rappresenta un livello di astrazione superiore che permette ad GAE di controllare l'allocazione delle risorse, l'elaborazione, la gestione delle richieste, lo scaling, e la distribuzione del carico di lavoro senza che l'applicazione ne sia cosciente. L'applicazione non viene eseguita su questa o quella macchina, con questo o quel sistema operativo, ma in un ambiente che può essere visto come una nuvola.

2.1.3 Datastore

Le applicazioni web hanno di solito bisogno di salvare informazioni. Lo scenario tipico è quello di utilizzare uno o più server su cui eseguire l'applicazione e un database centralizzato per l'intero sistema. Utilizzare un unico server per raccogliere i dati rende semplice avere un'unica rappresentazione degli stessi, in modo da permettere agli utenti di recuperare le stesse informazioni anche da server diversi. Ma un server centralizzato non è scalabile, e può diventare un collo di bottiglia una volta che ha raggiunto un certo limite di richieste.

GAE si discosta dal sistema centralizzato, ma anche dal modello relazionale, fornendo alle applicazioni un datastore distribuito basato su *Entities* e *Properties*.

I dati su GAE vengono salvati in entità. Ogni entità ha una o più proprietà, ognuna con nome e un valore che può essere di uno dei tipi di dato primitivi supportati. Ogni entità appartiene ad un determinato tipo (*kind*).

Apparentemente il modello non si discosta molto dal sistema relazionale, le entità di un particolare tipo possono essere viste come righe di una tabella, e le proprietà come le colonne. Ma ci sono due grandi differenze tra i due modelli. In primo luogo, non è necessario che entità di uno stesso tipo abbiano le stesse proprietà. Secondariamente, due entità possono avere una proprietà con uno stesso nome, ma valori di tipi differenti. Le entità possono quindi essere definite *schemaless*.

Un'entità può eventualmente designare un'altra entità come genitore (*parent*), rendendo così possibile organizzare le entità in una struttura ad albero, in cui l'entità senza genitore rappresenta la radice (*root entity*), mentre le entità figlie, le figlie delle figlie, e così via, rappresentano i suoi discendenti. Le entità discendenti da un medesimo ramo dell'albero, ovvero da una medesima entità *ancestor*, appartengono ad un *entity group*.

Ogni entità ha una chiave univoca (*key*) che, diversamente da come accade nei database relazionali, non è una sua proprietà, ma un suo aspetto indipendente. Una chiave è formata dai seguenti componenti:

- Il *kind* dell'entità
- Un identificatore, che può essere una stringa o un valore numerico, e che può essere definito dall'utente o generato automaticamente da GAE
- Eventualmente l'*ancestor path*, che determina la posizione dell'entità nell'albero dei discendenti della *root entity*

La chiave è utilizzata per determinare la posizione dell'entità nel vasto insieme di server che costituisce il datastore, per questo motivo una volta determinata non può essere cambiata.

Query

Mentre nei database relazionali le query sono eseguite in tempo reale su un insieme di tabelle, e possono essere velocizzate indicizzando una o più colonne, in GAE ogni query è associata ad un indice. Quando un'applicazione esegue una query, il datastore trova l'indice di quella query, lo scansiona finché non individua un'entità corrispondente ai criteri, e restituisce tutte le entità consecutive, fino all'ultima che soddisfa l'interrogazione.

Ciò richiedere naturalmente che GAE conosca in anticipo che tipo di query l'applicazione potrebbe eseguire, ovvero il tipo di entità, le proprietà interessate, gli operatori dei filtri, e l'ordine degli ordinamenti. GAE mantiene autonomamente indici per le query più comuni, basate sui filtri per tipo di entità o per proprietà, ma gli indici per query più complesse devono essere definite manualmente nella configurazione dell'applicazione.

Quando l'applicazione crea o modifica entità, il datastore si occupa quindi di aggiornare tutti gli indici. Ciò rende l'aggiornamento più dispendioso, ma riduce notevolmente la complessità computazionale delle query. Infatti sulle prestazioni non incide il numero delle entità memorizzate, ma solo la dimensione dell'insieme dei risultati.

Una query può anche includere un *ancestor filter*, ovvero limitare i risultati all'entity group di una specifica entità ancestor, tali interrogazioni sono dette ancestor query. Questo dettaglio è importante perché GAE garantisce che i risultati di una ancestor query siano sempre aggiornati all'ultimo cambiamento avvenuto, mentre ai risultati di una generica query potrebbero mancare aggiornamenti avvenuti negli ultimi instanti di tempo antecedenti all'operazione.

Transazioni

Così come avviene nei database relazionali, le operazioni effettuate sul datastore di GAE vengono eseguite in transazioni. Un'applicazione può leggere o aggiornare diverse entità in una singola transazione, ma è necessario che specifichi su quali di esse agisce. Qui entrano in gioco gli *entity group*, infatti GAE garantisce automaticamente le proprietà transazionali

se le entità involte fanno parte dello stesso gruppo, ciò è dovuto alla loro vicinanza fisica nel datastore. Transazioni che involvono più gruppi sono permesse ma devono essere dichiarate esplicitamente come *cross-group transaction*, e possono agire su un massimo di cinque gruppi.

Per la gestione della concorrenza, il datastore utilizza l'*optimistic concurrency control*, ovvero ogni utente è "ottimista" sul fatto che la transazione abbia successo, quindi non viene piazzato alcun lock sui dati. Quando una transazione effettua un commit, GAE controlla quando è avvenuto l'ultimo aggiornamento per quel gruppo, se è posteriore all'istante in cui la transazione ha effettuato la sua lettura questa fallisce.

In un tipico scenario in cui due utenti devono leggere e aggiornare gli stessi dati contemporaneamente, ciò che accade è il seguente:

1. il primo utente inizia la sua transazione leggendo i dati, l'operazione di lettura avviene all'istante di tempo t_1
2. il secondo utente inizia la sua transazione leggendo i dati, l'operazione di lettura avviene all'istante di tempo t_2
3. il primo utente aggiorna i dati, la sua transazione viene completata con successo all'istante t_3
4. il secondo utente tenta l'aggiornamento ma fallisce perché l'ultimo aggiornamento del gruppo è posteriore all'istante in cui la transazione corrente ha effettuato la lettura ($t_3 > t_2$)

Con questo meccanismo, nella maggior parte dei casi una transazione termina con successo. Ma se un'applicazione è progettata in modo che molti utenti possano aggiornare una singola entità, le probabilità di fallimento crescono. È importante definire accuratamente i gruppi di entità in modo da evitare il rischio anche con un vasto bacino di utenza.

2.1.4 Servizi

Oltre all'ambiente di esecuzione e al datastore, GAE mette a disposizione, tramite API, una serie di servizi utili alle applicazioni web.

Molte applicazioni web hanno risorse che non cambiano durante l'esecuzione, come pagine HTML statiche, CSS o immagini. Siccome accede a tali file non necessita l'esecuzione di codice dell'applicazione sarebbe inefficiente inoltrare le richieste alle

istanze o addirittura crearne di nuove. Invece GAE mette a disposizione degli appositi server su cui dirottare le richieste di contenuti statici.

La memcache è un servizio di storage chiave-valore che utilizza la memoria principale anziché il disco, quindi è estremamente più veloce del datastore. È un servizio distribuito, ogni istanza di un'applicazione ha perciò accesso allo stesso set di dati. Tuttavia non è garantito essere persistente, perciò il suo tipico utilizzo consiste nel caching dei risultati di una query.

Per memorizzare file di grandi dimensioni invece, GAE fornisce un servizio di storage alternativo, il Blobstore, che permette l'upload diretto di file attraverso una richiesta HTTP e l'accesso ai contenuti mediante interfacce standard di stream dei dati.

GAE può accedere a risorse web esterne grazio all'URL Fetch service. Il servizio effettua richieste HTTP ad altri server nella rete, in modo da permettere alle applicazioni di recuperare dati da altri siti web o di interagire con altri web service. Siccome i server esterni possono impiegare un tempo considerevole a rispondere, l'URL Fetch API può eseguire la richiesta in background, ma anche in questo caso la risposta deve pervenire nel tempo concesso all'applicazione.

Un'applicazione può mandare e ricevere messaggi grazie al Mail service. (per le email) e all'XMPP service. XMPP è un protocollo di chat e grazie al servizio omonimo l'applicazione può comunicare con qualsiasi client che lo supporta.

Infine lo user service permette agli utenti dell'applicazione di autenticarsi col loro Google Account, evitando di implementare un sistema di autenticazione ad-hoc per l'applicazione.

2.1.5 Task Queue e Cron Job

Gli standard di HTTP prevedono che un server dia risposta entro pochi secondi, quindi un'applicazione web non dovrebbe eseguire lunghe computazioni nel gestire la richiesta. Su GAE ciò diventa un obbligo, dato il limite di tempo di risposta di 60 secondi. Tuttavia GAE permette di eseguire operazioni complesse definendo dei task, ovvero unità di lavoro, che verranno eseguiti automaticamente, all'infuori della gestione di una richiesta HTTP.

I task possono essere messi in coda, per essere eseguiti in sequenza quanto prima, in questo caso si parla di Task Queue; oppure possono essere schedulati per essere eseguiti in un determinato istante di tempo, in questo caso si parla di Scheduled Task o Cron Job.

Un record di una task queue consiste in una richiesta HTTP, che GAE si occupa di gestire quando viene estratta dalla coda. Un task viene quindi gestito dall'applicazione allo stesso modo di una normale richiesta, ma con un'importante differenza, il limite di tempo di esecuzione è di 10 minuti. Se un task viene elaborato con successo viene eliminato dalla coda, altrimenti GAE ritenta finché l'elaborazione non termina correttamente. GAE mette a disposizione una coda di default con un rate di cinque task al secondo (vengono eseguiti al massimo cinque task in un secondo), ma nella configurazione di un'applicazione possono essere definite ulteriori code con un rate differente. I task possono essere inseriti nella coda durante la gestione di una normale richiesta tramite le apposite API.

2.1.6 App Engine Instance e Back-end

GAE riesce a scalare la potenza di calcolo disponibile utilizzando diverse istanze dell'applicazione, le quali vengono create on-demand automaticamente per soddisfare le richieste e disattivate quando rimangono inattive. Le istanze non sono configurabili, ciascuna ha bassa potenza di calcolo e circa 128MB di memoria, inoltre non è possibile sapere quale istanza risponde ad una determinata richiesta.

Per necessità in cui è richiesta maggiore flessibilità e potenza di calcolo GAE permette di definire manualmente i back-end da utilizzare per la creazione delle istanze. Gli App Engine Back-end sono istanze dell'applicazione esenti dal limite del tempo di esecuzione e hanno accesso a maggiore memoria (fino a 1GB) e potenza di calcolo (fino a 4.8GHz). I back-end non scalano automaticamente in base alle richieste, ma è necessario specificare manualmente il numero di istanze per ciascun back-end (fino a 20).

2.1.7 Strumenti di sviluppo

GAE mette a disposizione tre diversi runtime environment: un ambiente Java, un ambiente Python, e un ambiente basato su un linguaggio sviluppato da Google, Go.

L'ambiente Java permette di eseguire applicazioni compilate per la versione 6 della Java Virtual Machine (JVM) tramite una versione modificata della stessa. Un'applicazione sviluppata in Java può accedere alle funzionalità di GAE utilizzando le API standard del linguaggio. Le richieste web vengono gestite da normali Java Servlet, mentre per l'accesso al datastore è possibile utilizzare JPA o JDO. Le restrizioni della sandbox sono

implementate nella JVM, ad esempio se l'applicazione tenta di aprire un socket o scrivere su file verrà lanciata un'eccezione. Ogni altra libreria Java che non vìola le restrizioni della sandbox può tranquillamente essere eseguita su GAE.

L'ambiente Python esegue applicazioni scritte per la versione 2.7 di Python, utilizzando una versione modificata di Cpython, l'interprete ufficiale del linguaggio. GAE invoca un'applicazione Python tramite WSGI, un'interfaccia standard ampiamente supportata. Un'applicazione può utilizzare gran parte delle librerie standard di Python, nonché diversi framework open source per applicazioni web come Django, web2py, Pyramid e Flask. GAE include anche un framework minimale, webapp.

Ognuno dei tre ambienti di esecuzione ha il suo Software Development Kit (SDK), scaricabile dal sito in base al proprio sistema operativo. Gli SDK forniti includono un web server capace di emulare il sandbox environment e tutti i servizi di GAE in locale sul proprio computer. Ogni SDK include anche tutte le API della piattaforma, nonché un tool per eseguire l'upload dell'applicazione. Una volta creati codice, file statici e file di configurazione è sufficiente caricare il software tramite l'apposito strumento, il quale richiederà le credenziali di accesso del Google Account dell'utente.

Per l'amministrazione del software è anche possibile utilizzare una console di amministrazione web-based, la quale può essere utilizzata per creare nuove applicazioni, configurare domini, cambiare versione di un'applicazione, esaminarne accessi, datastore e log di errori. Inoltre è possibile invitare altre persone a partecipare allo sviluppo, consentendone l'accesso alla console e la possibilità di caricare nuove versioni.

2.2 Java Persistence

Java è un linguaggio ad oggetti, ogni applicazione Java è cioè composta da oggetti, i quali incapsulano dati. Spesso è necessario salvare tali dati, sia per risparmiare memoria che per preservarli dopo il termine dell'applicazione. I dati possono essere salvati utilizzando diverse tecnologie e supporti, genericamente definiti *data source*, in primis database e filesystem.

Le tecniche di accesso ai dati sono differenti per ogni tipo di *data source*, pertanto i

programmatori dovrebbero conoscere diverse API e in alcuni casi diversi linguaggi. Spesso è inoltre molto complesso mappare oggetti in determinati *data source*, argomento molto discusso è ad esempio l'*object-relational mapping* (ORM), ovvero la persistenza su database relazionali. Un pattern piuttosto diffuso è perciò il *persistence layer*, la realizzazione di uno strato software adibito alla persistenza degli oggetti, che permette alla logica dell'applicazione di accedere al *data source* tramite una semplice interfaccia, nascondo la complessità del processo di persistenza.

Il vantaggio del *persistence layer* è quello di rendere la logica dell'applicazione completamente indipendente dal meccanismo utilizzato per lo storage dei dati, permettendo agli sviluppatori di utilizzare liberamente classi Java per rappresentare le informazioni necessarie.

Java ha fatto della persistenza uno standard, esistono infatti due diverse specifiche che definiscono una generica API di persistenza: Java Persistence API (JPA) a Java Data Objects (JDO)[16]. Le caratteristiche di base delle due specifiche sono molto simili:

- Permettono di definire il mapping tra modello ad oggetti e data source tramite una specifica XML o annotazioni Java, in modo da non forzare l'utilizzo di determinati nomi o relazioni di ereditarietà nella realizzazione delle classi Java. Il mapping riflette nel data source i data type degli attributi degli oggetti e le relazioni tra le classi.
- Includono la specifica di un apposito linguaggio con cui interrogare il data source.
- Utilizzano una singola interfaccia come gateway.

Esistono però alcune differenze sostanziali. JPA è pensato prevalentemente come specifica per Object Relational Mapping. Sebbene esistano diverse implementazioni della specifica che utilizzano data source non relazionali, i predicati che definiscono il mapping si rifanno direttamente al gergo dei database (One-to-One, One-to-Many). Inoltre JPA, oltre a mettere a disposizione il proprio query language (JPQL), permette di comporre con un approccio ad oggetti (Criteria API) una query relazionale. L'approccio di JDO d'altro canto è più generale ed aderisce in modo più stretto al concetto di persistenza trasparente.

Esistono numerosi framework che implementano le specifiche precedentemente descritte. Tra i più utilizzati vi sono l'ORM Hibernate[7] e DataNucleus Access Platform[8].

7 http://www.hibernate.org/
8 http://www.datanucleus.org

	JPA	JDO
Java Package	`javax.persistence`	`javax.jdo`
Persistent class definition	`Entity`	`PersistenceCapable`
Gateway Interface	`EntityManager`	`PersistenceManager`
Query	*JPQL, Criteria API*	*JDOQL*

Table 1: JPA and JDO API overview

2.3 Restlet Framework

Restlet è un framework open source per la realizzazione di applicazioni Java conformi ai principi di REST.[17]

La caratteristica principale di Restlet consiste nell'offrire una libreria Java compatta e portabile che incorpora i più rilevanti concetti di REST, vale a dire:

Uniform interface

Metodo standard per interagire con risorse tramite richieste e risposte.

Component

Contenitori logici di applicazioni Restlet.

Connector

Abilitano la comunicazione tra componenti utilizzando uno specifico protocollo.

Representation

Espongono lo stato di una risorsa.

2.3.1 Architettura

Restlet è composto da un core snello e un ricco set di estensioni. Il core è a sua volta composto da Restlet Engine, in cui risiede la logica vera e propria, e dalle Restlet API, che ne espongono le funzioni agli sviluppatori.

Le estensioni aggiungono supporto ai vari tipi di dati utilizzabili nelle rappresentazioni (XML, JSON, HTML, XSLT, ecc.), connettori addizionali (POP3, SMTP, FTP, ecc.) o Plug-in di terze parti (Apache FileUpload, FreeMarker, Jackson, JAXB, ecc.).

Restlet identifica un singolo host fisico come `Component`, il quale può ospitare una o più `Application` e avere uno o più `Connector` per comunicare con l'esterno. Un `Connector` può essere un `Client` o un `Server`, identificando il ruolo del componente nella specifica comunicazione.

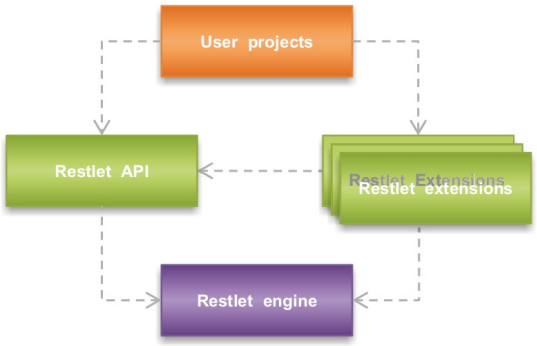

Figure 5: Overall Restlet Design

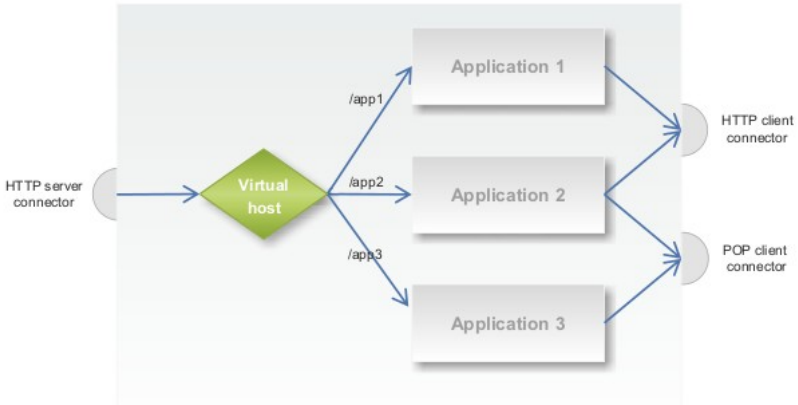

Figure 6: Restlet Component

2.3.2 Sviluppo di un'applicazione con Restlet

La classe `Application` di Restlet permette di implementare una RESTful API raggruppando risorse (implementate mediante la classe `Resource`) che condividono gli stessi dati e servizi. Restlet lascia piena libertà sull'implementazione di *presentation* e *persistence layer*, per i quali il programmatore può utilizzare qualunque altro framework, direttamente o attraverso estensioni.

Qualunque richiesta diretta ad una particolare risorsa viene gestita dall'`Application`. La richiesta viene prima processata da un livello di *filtering*, che può gestire aspetti come compressione dei dati e autenticazione. Successivamente la chiamata passa al *routing layer*, che ne determina la destinazione finale, tipicamente in base all'URI della richiesta. Al termine la chiamata può raggiungere il *resource handling layer*, in cui la `Resource` si occupa di gestire la richiesta ed elaborare una risposta da recapitare al client. Non essendo Restlet limitato all'implementazione del server, l'`Application` gestisce allo stesso modo richieste generate internamente, processandole in modo inverso.

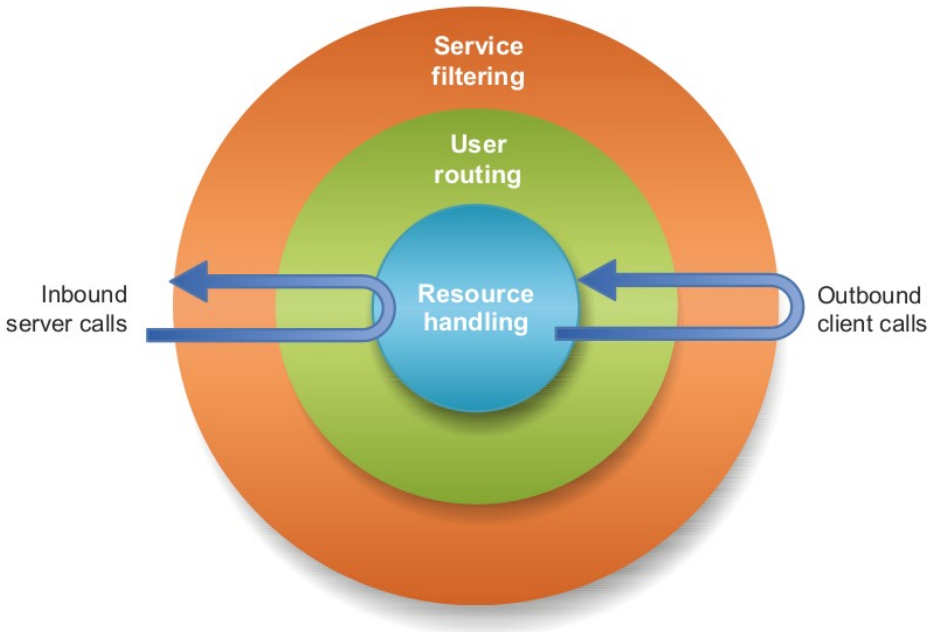

Figure 7: Restlet Application Layers

2.3.3 Restlet Resource

La classe `Resource` di Restlet è progettata per essere il più analoga possibile al concetto di risorsa in REST. Ad ogni richiesta il Restlet Engine crea una nuova istanza della `Resource` richiesta, la quale rappresenta lo stato attuale della risorsa. Lo stato può essere letto o manipolato attraverso i metodi standard di HTTP (`GET`, `PUT`, `DELETE`, e `POST`). Il comportamento della `Resource` su client e server è differente, per questo nella pratica vengono utilizzate le sottoclassi `ClientResource` e `ServerResource`.

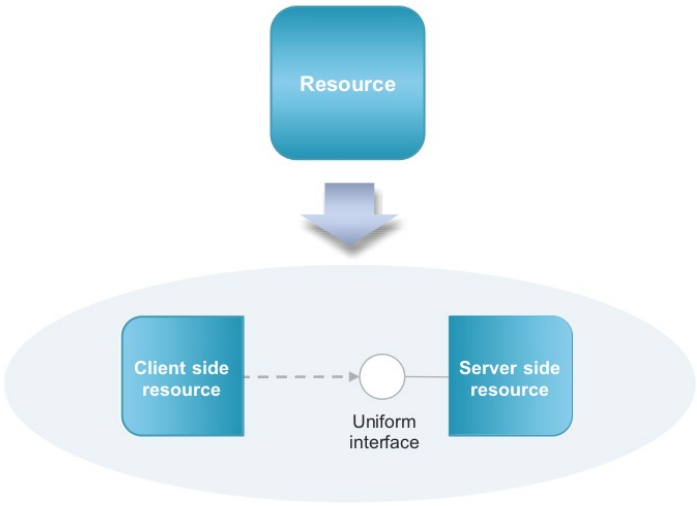

Figure 8: Decomposition ofan abstract resource into Restlet artifacts

2.3.4 Restlet Representation

L'interazione tra `ClientResource` e `ServerResource` consiste nello scambio di rappresentazioni delle risorse, per farlo Restlet utilizza la classe `Representation`. `Representation` può virtualmente rappresentare qualunque tipo di dato, conforme all'internet media type[9] e specificato dall'attributo `MediaType`. Il contenuto di un'istanza di `Representation` può essere letto come stream di Java ma Restlet mette a disposizione una serie di sottoclassi (e molte altre sono disponibili nelle estensioni) per recuperare le informazioni necessarie tramite metodi Java.

9 http://www.iana.org/assignments/media-types

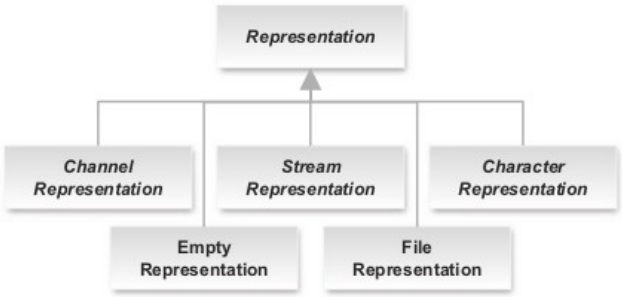

Figure 9: Subclasses of Representation

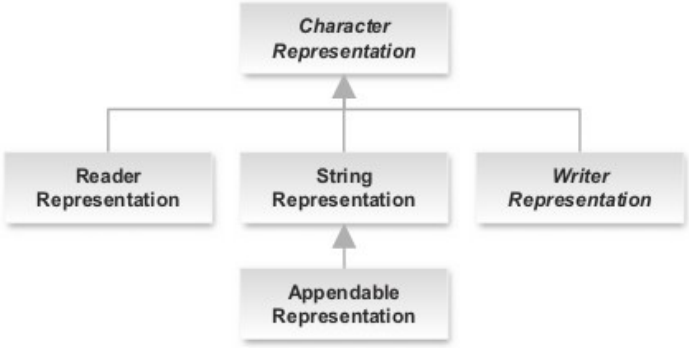

Figure 10: Character-basedrepresentation classes

2.3.5 Astrazione di HTTP

Una delle caratteristiche fondamentali di Restlet è quella di lavorare direttamente con HTTP (e potenzialmente con altri protocolli applicativi). Sebbene possa sembrare un aspetto negativo quello di avere a che fare con il protocollo, HTTP è a livello applicativo e supporta caratteristiche avanzate come caching, richieste condizionali, negoziazione dei contenuti tra diversi formati e lingue, e il web è costruito direttamente su di esso.

Per facilitare l'utilizzo di HTTP, Restlet astrae ed espone, in un organico modello ad oggetti, tutta la semantica a livello applicativo del protocollo, mappando header e status code in classi Java.

2.3.6 Realizzazione di un web service

Un web service è essenzialmente una web application, cioè un'applicazione lato server che utilizza HTTP come protocollo di comunicazione. In termini di Restlet API ciò si traduce in un `Component` con un `Server` connector con supporto HTTP.

Come detto in precedenza una WEB API consiste in un insieme di risorse. Per ogni risorsa del web service bisogna estendere la classe `ServerResource` per gestirne l'handling delle richieste. Una Restlet Application è associata ad un endpoint, l'handler è determinato dal routing layer in base al path.

Una `ServerResource` deve gestire uno o più metodi HTTP, ognuno dei quali è mappato ad un omonimo metodo Java della suddetta classe, che lo sviluppatore può modificare tramite override. Tuttavia il sistema consigliato per l'implementazione del resource handling consiste nel creare nuovi metodi ad-hoc e marcarli con annotazioni Java (`@Get, @Put, @Post, @Delete`), per mappare un metodo HTTP al metodo annotato.

Il grande vantaggio di utilizzare le annotazioni consiste nella libertà di utilizzare qualunque tipo ti dato come argomento e come tipo di ritorno del metodo annotato, lasciando la conversione da e verso la classe `Representation` a Restlet. In questo modo, oltre a permettere al programmatore di focalizzarsi solo sulla logica del metodo si ha la possibilità di usufruire della content negotiation di HTTP. A seconda del client, un generico Java bean può essere ad esempio trasferito come oggetto Java serializzato, convertito in XML (grazie all'estensione per JAXB) o in JSON (grazie all'estensione per Jackson).

Restlet permette di eseguire un web service sia in un ambiente JavaSE che in un Servlet Container (JavaEE). Nel primo caso è necessario utilizzare un `Component` e un `Server` connector che funga da web server, per un'installazione robusta Restlet mette a disposizione un'estensione che consente di utilizzare Jetty, il più popolare web server scritto in Java. Per un ambiente JavaEE invece Restlet fornisce un'apposita servlet, `ServerServlet`, a cui agganciare l'`Application`.

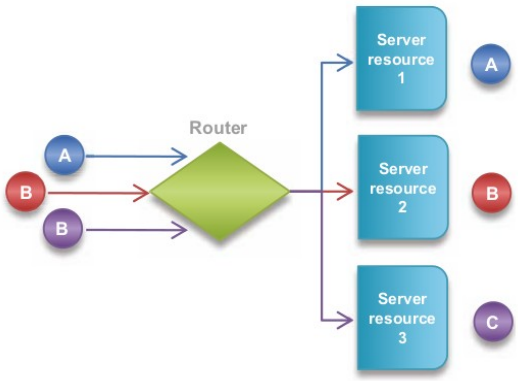

Figure 11: Router distpatching calls

2.4 XML Data Binding

Essendo XML lo standard più diffuso di rappresentazione delle informazioni, spesso i programmatori devono affrontare il problema di manipolare dati espressi in tale linguaggio. Esistono due API utilizzati in genere per accedere ad un documento XML: Domain Object Model (DOM) o Simple API for XML (SAX).[18]

DOM rappresenta un documento XML in una struttura dati ad albero, che può essere navigata ed esaminata dall'applicazione. SAX invece è una API ad eventi (event-driven): l'applicazione registra il suo interesse in determinati eventi, come l'inizio di un tag, attributo o testo, i quali vengono lanciati durante il parsing del documento. La differenza principale tra le due risiede nel fatto che DOM legge e carica preventivamente in memoria l'intero documento, mentre SAX restituisce i dati progressivamente durante il parsing.

In entrambi i casi il programmatore ha comunque a che fare con una mera rappresentazione della struttura XML, operando attraverso generici elementi, attributi e testo. Al crescere della struttura di un documento il programmatore è quindi costretto a scrivere una considerevole quantità di codice di bridging, che identifica e trasforma informazioni espresse in XML in rappresentazioni più adatte all'elaborazione attraverso la logica dell'applicazione.

Table 2: Data type bindings

XML Schema Type	Java Data Type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:positiveInteger	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:unsignedLong	java.math.BigDecimal
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType (for xsd:element of this type)	java.lang.Object
xsd:anySimpleType (for xsd:attribute of this type)	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

L'XML Data Binding è un approccio moderno all'XML processing, e si basa sull'idea di ricavare da un documento XML un modello ad oggetti (e vice versa). Il grande vantaggio consiste nell'offrire direttamente al programmatore una serie di oggetti che possono essere utilizzati nella logica dell'applicazione, evitando il passaggio di traduzione delle informazioni.

JAXB

Esistono diversi strumenti di XML Data Binding per ogni linguaggio di programmazione object-oriented, quello utilizzato nel lavoro di tesi è JAXB (Java Architecture for XML Binding), incluso direttamente nel JDK a partire dalla versione 1.6. JAXB fornisce due principali funzionalità: permette di convertire oggetti Java in XML, operazione nota come *marshalling*; e di ricavare oggetti Java da un documento XML, ovvero *unmarshalling*. Le classi Java rappresentanti il modello XML possono essere generati automaticamente da un XML Schema grazie al tool `xjc`, anch'esso incluso nel JDK. Le classi risultanti includono annotazioni Java appartenenti al namespace `javax.xml.bind.annotation.*`: l'annotazione `@XmlRootElement` indica che la classe annotata è un elemento radice di un documento XML. Gli attributi della classe radice sono invece marcati come `@XmlElement`, e il loro tipo viene definito mappando i data type dell'XML Schema in base alla tabella 1. Gli elementi di tipo `xs:complexType` invece vengono mappati in altre classi opportunamente generate, mentre elementi che possono occorrere più di una volta vengono inclusi in attributi di tipo `java.util.List`.

In alternativa JAXB permette anche di generare l'XML Schema a partire da classi annotate grazie al tool `schemagen`. A tale scopo le annotazioni permettono di specificare alcuni attributi come nome elemento o occorrenze.

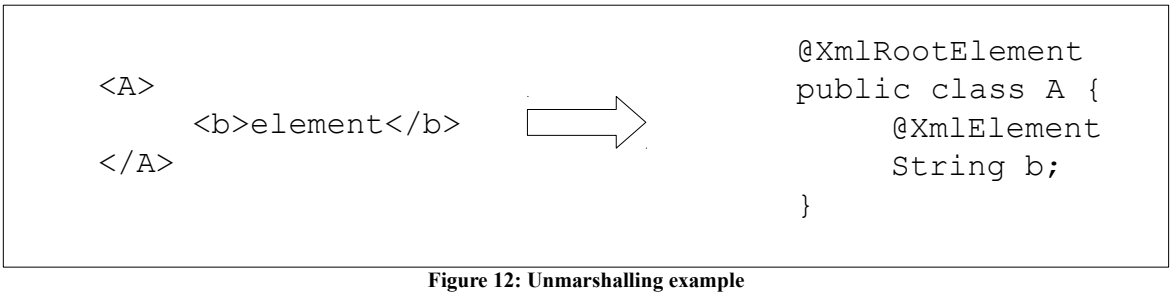

Figure 12: Unmarshalling example

XPath

Altro metodo di estrapolazione di informazioni da XML è invece XPath, un linguaggio definito dal W3C specifico per effettuare query all'interno del documento XML. XPath permette di navigare all'interno dell'albero definito dal documento XML e di selezionare nodi in base ad una varietà di criteri. Grazie a XPath è possibile recuperare direttamente le

informazioni necessarie dal documento, tralasciando dati superflui ed evitando di processare l'intero codice.

Una query XPath (comunemente definita espressione XPath, o più semplicemente XPath) consiste in un percorso (location path) composto da una sequenza di passi (location step) separati da slash (/). Ciascuno step è formato da tre elementi:

- *Axis specifier* Specifica la posizione di un nodo rispetto alla radice del percorso o allo step precedente
- *Node test* Identifica un nodo
- *Predicato* Filtra i nodi identificati

La sintassi di un location step è la seguente:

```
axisname::nodetest[predicate]
```

Gli axis specifier disponibili sono elencati in tabella 3

Table 3: Available axes

AxisName	Result
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself
attribute	Selects all attributes of the current node
child	Selects all children of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself
following	Selects everything in the document after the closing tag of the current node
following-sibling	Selects all siblings after the current node
namespace	Selects all namespace nodes of the current node
parent	Selects the parent of the current node
preceding	Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

Un node test è tipicamente il nome dei nodi da selezionare, ma può anche essere sostituito dal carattere wildcard '*'. Tra i nodi individuati tramite percorso e test, è possibile poi selezionare quelli che soddisfano particolari condizioni tramite predicati. I predicati possono ad esempio includere filtri sugli attributi, o indici che individuano un particolare nodo nel vettore dei risultati.

I costrutti più comuni sono inoltre esprimibili in notazione ridotta:

- `child::` può essere omesso
- `@name` seleziona un attributo col nome specificato (equivalente allo step `attribute::name`)
- `.` (punto) è equivalente allo step `self::*`
- `..` (punto-punto) è equivalente allo step `parent::*`
- `//` è equivalente a `/descendant-or-self::*/`

Un semplice esempio di espressione XPath è la seguente:

```
/A/B/C
```

che seleziona tutti i nodi C figli dei nodi B figli dell'elemento radice A. Invece

```
A//B/*[1]
```

Seleziona il primo nodo figlio ('[1]'), di qualsiasi nome ('*') di un elemento B discendente di un generico nodo A; l'espressione può restituire più di un risultato.

2.5 JxReport

jxReport è una libreria Java open source realizzata come progetto correlato al presente lavoro di tesi[19]. JxReport è capace di creare Report in HTML avendo in input un file XML . Il documento XML è strutturato a partire dall'elemento radice report, il quale contiene una sequenza di elementi che devono essere rappresentati nell'output.

La libreria può essere utilizzata in qualunque ambiente supporti una Java Virtual

Machine. L'output HTML è estremamente portabile, tale codice può essere inserito all'interno di un file, inglobato in un Android WebView, iniettato in una JSP, ecc.

Un report contiene una sequenza di oggetti, ognuno dei quali ha un tipo e un insieme di attributi. I tipi attualmente supportati sono tabella, testo, diagramma a torta, diagramma a barre, diagramma a linee. Gli attributi di un oggetto variano a seconda del suo tipo, tuttavia sono previsti quattro attributi comuni a tutte le tipologie:

Id

Identifica univocamente un oggetto all'interno del report

Class

Utilizzato per raggruppare una famiglia di oggetti che condividono una serie di proprietà

Titolo

Una stringa che sarà visualizzata al di sopra dell'oggetto

Regole CSS

Una lista di regole CSS da applicare direttamente al codice HTML risultante. L'utilizzo dei CSS rende l'output grafico estremamente personalizzabile, permettendo di cambiare font, dimensione dei caratteri, padding e margini.

I dati possono essere inseriti direttamente nel documento XML, tra gli attributi di un determinato oggetto; oppure possono essere sostituiti da particolari marcatori e letti da un file CSV esterno, durante la generazione dell'output. Tale caratteristica permette di definire un report a prescindere dai dati che rappresenta (*model*), in modo da utilizzare la stessa struttura (*view*) per diversi modelli.

```
<report>
  <barChart>
. . .
  </barChart>
  <table>
. . .
  </table>
  <textContent>
. . .
  </textContent>
  <lineChart>
. . .
  </lineChart>
  <pieChart>
. . .
  </pieChart>
</report>
```

Snippet 2: jxReport XML example

2.6 OAuth

OAuth è un protocollo che consente ad applicazioni di terze parti di ottenere accesso limitato ad un servizio HTTP, in vece di un utente detentore della risorsa, attraverso una procedura di approvazione interattiva tra l'utente e il servizio.[20]

Il protocollo è stato ideato per ovviare ad alcuni limiti del tradizionale modello di autenticazione client-server, che non permette di abilitare l'accesso di un'applicazione di terze parti ad una risorsa protetta se non condividendo le credenziali di autenticazione con questi. Condividere le credenziali infatti pone diversi problemi e limitazioni:

- Le applicazioni di terze parti devono memorizzare le credenziali dell'utente per usi futuri, tipicamente si tratta di una password in chiaro.
- I server devono supportare l'autenticazione con password, nonostante le inerenti vulnerabilità di questo sistema.

- Le applicazioni di terze parti ottengono accesso a tutte le risorse dell'utente sul server su cui hanno accesso. L'utente non ha la possibilità di restringere l'accesso ad un insieme limitato di risorse né di limitarne il tempo di utilizzo.
- L'utente non ha possibilità di revocare l'accesso ad una particolare applicazione senza revocarlo a tutte le altre, dato che l'unico modo per farlo è cambiando la password.
- Se una delle applicazioni di terze parti è compromessa allora lo sarà anche la password dell'utente e tutti i dati associati.

OAuth permette di evitare i problemi elencati aggiungendo un livello di autenticazione e separando il ruolo del client dal server detentore della risorsa. In OAuth, il client richiede accesso a risorse controllate da un utente (*resource owner*) e ospitate su un server (*resource server*), utilizzando credenziali diverse da quelle del proprietario.

Invece di utilizzare le credenziali dell'utente, il client ottiene un *access token*, ovvero una stringa che specifica risorse, durata e altri attributi di accesso. L'*access token* è rilasciato a client di terze parti da un server di autorizzazione (*authorization server*) con l'approvazione del *resource owner*. Il client utilizza l'*access token* per accedere alle risorse protette ospitate sul *resource server*.

Ad esempio un utente (*resource owner*) può consentire ad un server di stampa (*client*) di accedere alle sue foto protette localizzate su un servizio di photo-sharing (*resource server*). Anziché condividere username e password con servizio di stampa, l'utente si autentica sull'*authorization server* del servizio di photo-sharing, il quale genera specifiche credenziali (*access token*) per la delega al servizio di stampa.

2.6.1 Attori

OAuth definisce quattro attori:

Resource Owner

Un'entità che ha accesso ad una risorsa protetta.

Resource Server

Il server su cui sono localizzate le risorse protette, capace di accettare richieste a risorse protette autorizzate tramite *access token*.

Client

Un'applicazione che richiede accesso a risorse protette per conto del *resource owner* e con la sua autorizzazione. OAuth non richiede alcuna particolare implementazione del client, esso può ad esempio essere un'applicazione web, desktop o mobile.

Authorization Server

Il server che emette *access token* al *client* dopo l'autenticazione e l'autorizzazione del *resource owner*.

2.6.2 Access Token

Gli access token sono credenziali utilizzabili per accedere a risorse protette. Un access token è una stringa che rappresenta un'autorizzazione concessa al client. OAuth non pone alcun vincolo sul formato degli access token, essi possono avere diversi formati, strutture, metodi di utilizzo, proprietà crittografiche, in base ai requisiti di sicurezza del resource server. Tuttavia un token deve avere due parametri fondamentali: *lifetime* e *scope*.

Lo *scope* individua una singola risorsa a cui il client ha intenzione di accedere, o più informalmente individua una singola azione che il client eseguirà. Il resource server d'altro canto impedirà di eseguire qualunque altra operazione non presente negli scope. Prendendo ad esempio Google come resource server, alcuni possibili scope potrebbero essere inserire eventi nel calendario, accedere alla rubrica, leggere i file su Google Drive, ecc.

Il *lifetime* invece indica il tempo di validità del token. Per questioni di sicurezza infatti il token ha una scadenza relativamente breve, il protocollo non specifica la durata ma nella pratica non supera i 30 minuti. Per ottenere all'occorrenza un nuovo access token il client può utilizzare un *refresh token*.

I *refresh token* sono credenziali utilizzabili per ottenere access token. I *refresh token* sono rilasciati anch'essi dall'authorization server al client. Il rilascio di questi è opzionale e a discrezione dell'authorization server. Se l'authorization server rilascia un *refresh token* lo include nella procedura di rilascio dell'access token.

2.6.3 Sequenza di autorizzazione

A. Il client richiede l'autorizzazione al resource owner. Ciò può essere fatto direttamente dal client stesso qualora sia eseguito sullo stesso dispositivo dell'utente, oppure tramite l'interazione con l'authorization server nel caso il client sia una web application. Nel secondo caso tipicamente l'utente interagisce con client tramite un browser web e la richiesta viene effettuata grazie ad un redirect in un'apposita pagina web dell'authorization server. L'authorization server richiede l'autenticazione dell'utente e successivamente mostra a questi i dettagli sull'applicazione e gli scope che essa richiede, a questo punto l'utente può concedere o meno l'autorizzazione.

B. Il client riceve l'autorizzazione, la quale può essere rappresentata in varie forme. Nel caso di applicazione nel medesimo dispositivo dell'utente può essere rappresenta da username o password, se invece l'authorization server fa da intermediario allora sarà rappresenta da un apposito codice (*authorization code*). In altri casi questa fase viene saltata e il client riceve direttamente l'access token; in questo caso si parla di *implicit grant* ed è ciò che avviene tipicamente in client Javascript.

C. Il client richiede l'access token autenticandosi con l'authorization server e consegnandogli la sua autorizzazione.

D. L'authorization server autentica il client e valida la sua autorizzazione. Se questa è valida consegna access token ed eventualmente refresh token.

E. Il client richiede di accedere alla risorsa protetta presentando l'access token. I metodi mediante i quali il token può essere trasmesso sono descritti in dettaglio nell'RFC6750.

F. Il resource server valida il token e, in caso di successo, elabora la richiesta.

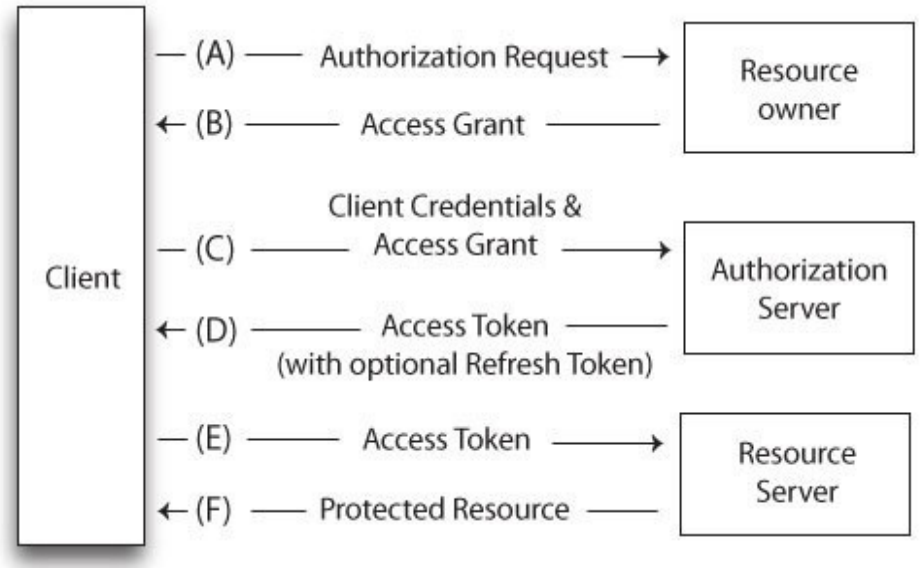

Figure 13: Protocol Flow

2.7 Maven

Maven è un build automation tool, principalmente utilizzato per la gestione di progetti Java. È pensato per assistere il programmatore in tutte le fasi del processo di sviluppo e rilascio di un progetto. Tale processo è esplicitamente definito da Maven come *build lifecycle*.

Così come il processo è diviso in varie fasi, il build lifecycle è suddiviso in *build phase:*

`validate`

Verifica che il progetto sia corretto e che tutte le informazioni siano disponibili.

`compile`

compila il codice sorgente del progetto.

`test`

Esegue il test del codice compilato utilizzando un apposito framework di unit testing. Non è necessario che il codice di test venga incluso nel pacchetto di rilascio del software.

`package`

Viene generato il pacchetto con cui il software viene rilasciato (es. Jar, War).

`integration-test`

Esegue il deploy del pacchetto in un ambiente di runtime dove è possibile eseguire dei test di integrazione.

`verify`

Esegue dei controlli per verificare la validità del pacchetto.

`install`

Installa il pacchetto nel repository che Maven mantiene localmente, in modo che possa essere usato da altri progetti.

`deploy`

Invia il pacchetto ad un repository remoto o ad un altro runtime environment in cui il software viene eseguito.

Le build phase vengono richiamate dal programmatore tramite il tool da riga di comando `mvn`:

```
mvn <phase>
```

La loro esecuzione è tuttavia sequenziale, perciò la chiamata di una build phase comporterà l'esecuzione di tutte le precedenti. Ad esempio con il comando

```
mvn deploy
```

vengono automaticamente eseguite tutte le build phase.

Tuttavia, anche se una build phase è responsabile di una fase del build lifecycle, il metodo con cui questa giunge a compimento può variare. Per questo una build phase contiene dei *goal*, ovvero delle specifiche operazioni (task) che contribuiscono allo

sviluppo e al mantenimento del progetto. Un goal può essere quindi associato ad una o più build phase, o anche a nessuna, in quest'ultimo caso il goal non fa parte del build lifecycle ma può comunque essere richiamato tramite mvn qualora il programmatore abbia bisogno di eseguirlo.

Altro obiettivo di Maven è fornire ai programmatori linee guida per una buona pratica di sviluppo. A tal proposito ogni progetto Maven ha tre proprietà:

`groupId`
: Identifica il software, come tale dovrebbe essere univoco, per questo è consigliato che sia un nome di dominio che si possiede (es. org.bodycloud). Il groupId inoltre dovrebbe essere usato come nome del package principale.

`artifactId`
: è il nome di un singolo modulo di un software, vale a dire il nome del progetto (es. bodycloud-server, bodycloud-engine).

`version`
: la versione dell'artifact (es. 2.0, 2.0.1, 1.3.1).

Inoltre Maven predefinisce la struttura delle directory da utilizzare. Il codice sorgente deve essere incluso nella cartella `src`, e diviso secondo lo schema in figura 14.

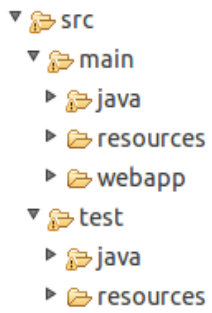

Figure 14: Maven artifact directory structure

Di seguito vengono descritti i dettagli relativi a ciascuna sotto-cartella:

`src/main/java`
: Contiene il codice sorgente Java

`src/main/resources`

Contiene risorse utili a runtime (es. file di configurazione), viene incluso nel classpath di Java

`src/main/webapp`

Utilizzata soltanto dalle web application. Contiene il web.xml, le cartelle WEB-INF e META-INF e le risorse web (es. pagine JSP, pagine HTML, Javascript e CSS)

`src/test/java`

Contiene il codice Java di unit testing. Viene inclusa nel classpath solo durante la fase di test.

`src/test/resources`

Risorse utilizzate durante gli unit test. Viene inclusa nel classpath solo durante la fase di test.

La configurazione di un progetto è inclusa in un singolo file XML, il Project Object Model (**pom.xml**). Nel pom vengono definiti in sequenza `groupId`, `artifactId`, `version`, dipendenze e build phase.

Le dipendenze sono anch'esse artifact, e come tali sono identificate dai rispettivi `groupId`, `artifactId` e `version`. La gestione delle dipendenze è un aspetto fondamentale del build lifecycle: Maven infatti si occupa di cercare e scaricare automaticamente le dipendenze quando richiesto da una build phase. La ricerca viene effettuata nel Maven Central repository, ma nel pom è possibile specificare repository aggiuntivi. Una volta scaricati, gli artifact vengono salvati in un repository locale, in modo da evitare di ripetere le ricerca in rete. Utilizzando Maven quindi, per aggiungere una libreria al progetto che si sta sviluppando è sufficiente aggiungere poche righe, inoltre se la libreria in questione ha a sua volta altre dipendenze verranno risolte automaticamente in cascata.

Le build phase invece sono specificate come plugin. L'architettura di Maven permette di utilizzare come plugin qualsiasi applicazione che sia controllabile tramite standard input, cosa che rende teoricamente possibile l'utilizzo di qualunque compilatore, unit test tool, runtime environment, ecc. Anche i plugin sono Maven artifact, perciò vengono identificati e inclusi come se fossero dipendenze.

Infine Maven permette la creazione di nuovi progetti a partire da template predefiniti,

gli archetype. Un archetype contiene tipicamente un pom preconfigurato e una base di codice come punto di partenza in modo da rendere possibile il setup di un progetto in pochi secondi. L'uso degli archetype risulta particolarmente utile quando si ha a che fare con framework che richiedono una configurazione preliminare, o con progetti che necessitano di un build lifecycle particolare.

3 Progettazione

Obiettivo della tesi è realizzare un'infrastruttura per il Bodycloud, integrando le BSN con una piattaforma cloud estendibile, in grado di supportare diverse sorgenti di dati e di offrire diversi servizi di analisi.

In particolare l'architettura è stata sviluppata tenendo conto dei seguenti requisiti:

- Offrire funzionalità per ricevere e gestire in modo generico diversi flussi di dati provenienti da BSN.
- Realizzare un framework che permetta a programmatori, analisti ed esperti di dominio di realizzare applicazioni (o *servizi*) da eseguire nel cloud, per specifici casi d'uso inerenti la natura dei dati e che includano processi di analisi e generazione di report.
- Permettere lo storage dei dati in modo da riutilizzare gli stessi in tempi diversi.
- Realizzare un server che offra un'interfaccia generica per l'interazione con client di terze parti, in particolare con applicazioni mobile, che comprenda l'invio dei dati (input) e la reportistica (output).
- Rendere disponibili i servizi cloud mediante tale interfaccia, in modo che la realizzazione di un nuovo servizio non implichi l'aggiornamento delle applicazioni client.

3.1 Problematiche

Di seguito vengono analizzate le principali problematiche riscontrate nella progettazione del sistema:

Gestione di dati eterogenei

Il sistema deve poter gestire dati eterogenei, che pertanto hanno un determinato formato, solitamente raggruppati in grandi quantità (*dataset*).

Storage di dati eterogenei

Il sistema deve prevedere lo storage di dataset arbitrari, è pertanto necessario utilizzare un data source o una sua astrazione appropriata. I database relazionali ad esempio, non si prestano bene a tale necessita che richiederebbe la creazione continua di nuove tabelle.

Approccio PaaS

Lato cloud il sistema si presta alla realizzazione come PaaS, ovvero come piattaforma su cui implementare end-user applications (SaaS). Tuttavia la realizzazione di un PaaS è piuttosto complessa, in quando sarebbe necessario gestire la scalabilità, il load balancing e il deploy delle applicazioni finali.

Separazione delle funzionalità

Un'applicazione bodycloud può essere rappresentata da un processo composto da tre fasi: input, analisi e reportistica. Si tratta di tre aspetti relativamente indipendenti, diverse applicazioni potrebbero infatti avere lo stesso input e/o lo stesso output. Sarebbe opportuno gestire i tre aspetti separatamente in modo da permettere il riutilizzo dei componenti.

Separazione dei ruoli

Il soggetto che indossa i sensori è solitamente diverso da chi analizza i dati, le applicazioni devono quindi avere due casi d'uso differenti.

Interfaccia aderente agli standard

Il sistema deve poter interagire con software di terze parti, è pertanto indispensabile utilizzare uno standard di comunicazione riconosciuto.

Estensibilità dei client

Deve essere possibile estendere dinamicamente le funzionalità di un client parallelamente con il cloud, senza apportare modifiche. Ciò sarebbe possibile ad esempio permettendo il download da parte dei client di codice relativo a nuove funzionalità, ma ne comprometterebbe l'interoperabilità.

3.2 Scelte effettuate

L'idea alla base della progettazione è quella di partire dallo stato attuale dell'arte del data analysis e di estenderla al cloud computing. In particolare si tratta di trasporre il concetto di workflow per l'analisi dei dati nella rete, identificando nei nodi periferici i dispositivi con cui gli utenti interagiscono col sistema, con un occhio di riguardo ai mobile device.

La parte centrale del sistema è un SaaS (*cloud-side* da qui in avanti), ovvero una singola applicazione cloud che può essere configurata opportunamente dagli utenti in modo da offrire nuovi servizi.

Il *cloud-side* è a sua volta ospitato da un PaaS, tra i vari PaaS, la scelta è ricaduta su Google App Engine che, oltre a essere il più popolare, permette di mettere online un'applicazione professionale gratuitamente e con minimo sforzo. Altro fattore rilevante nella scelta riguarda la possibilità di integrazione con altri servizi e prodotti Google, in particolare Android, piattaforma preferita per l'implementazione di client lato body.

Il linguaggio di programmazione scelto è Java, in quanto linguaggio ad alto livello per cui è possibile facilmente trovare una gran quantità di framework completi e librerie open source.

Il sistema prevede tre tipi di attori. Il primo è l'utente utilizzatore della BSN (*body*), colui che fornisce l'input, tramite sensori, al sistema. Il secondo è quello che riceve l'output dell'analisi (*viewer*), il quale può coincidere col primo oppure essere un ente esterno, un monitore interessato all'analisi dei dati provenienti dal body analizzati in un particolare dominio applicativo. Il terzo è un esperto di analisi (*data analyst*), il quale si occupa di modellare il processo di elaborazione dei dati, configurando opportunamente il SaaS e venendo condo a quelle che sono le necessità del viewer.

3.3 Architettura

Come mostrato in figura 15, l'architettura risultante è composta da quattro diverse parti:

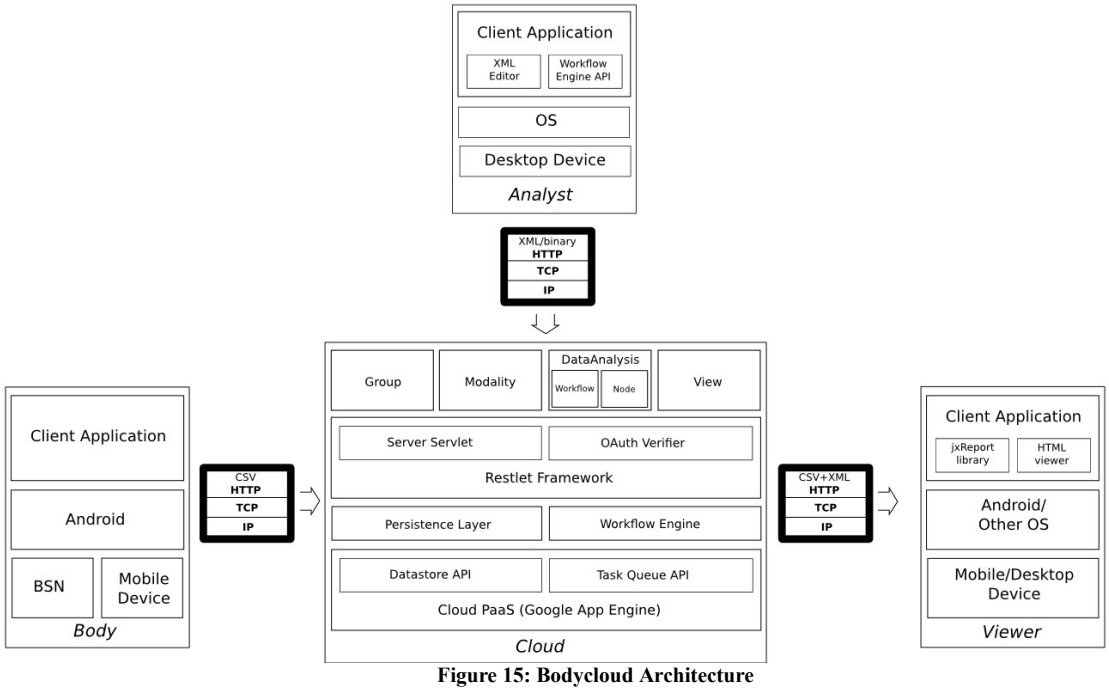

Figure 15: Bodycloud Architecture

Body

La parte del sistema adibita al monitoring di un soggetto attraverso sensori indossabili e che invia i dati raccolti da questi al cloud attraverso un dispositivo mobile. Applicazioni di questo tipo sono attualmente realizzabili su Android grazie al framework SPINE[21].

Viewer

La parte del sistema in grado di visualizzare l'output di un processo di analisi, attraverso un'opportuna libreria di reportistica come jxReport.

Cloud

La parte del sistema che si occupa dello storage e del processing dei dati e che offre supporto per la configurazione dei servizi e per la visualizzazione dei report.

Analyst

La parte del sistema che si occupa di creare nuovi servizi bodycloud.

3.4 Architettura del Cloud-side

Il cloud-side offre un'astrazione di alto livello per la configurazione e l'utilizzo dei servizi. Un servizio non è un'unica entità ma è definito in base a quattro diversi aspetti:

Workflow

Formalizza un processo di analisi basato sull'elaborazione sequenziale di un dataset attraverso la concatenazione di unità di elaborazione (nodi). Si rifà direttamente ai pipelining tool in quanto approccio largamente riconosciuto e che permette di utilizzare uno di questi strumenti per l'esecuzione dei processi di analisi

View

Formalizza una rappresentazione grafica di dati in output ad uso esclusivo del viewer. Tale specifica è pensata per separare quello che è il risultato di un processo di analisi (model) dalla sua rappresentazione grafica (view), permettendo l'utilizzo di diverse view per uno stesso model, e di una stessa view per model diversi. È possibile utilizzare qualsiasi linguaggio per creare una view, tuttavia il sistema supporta e promuove l'utilizzo di XML e jxReport, libreria realizzata come progetto correlato appositamente per questo scopo.

Group

Identifica il servizio e l'organizzazione o ente che lo offre. I dati inviati dal *body* al *cloud* devono essere associati ad un gruppo. Mediante i gruppi gli utenti sono coscienti dell'entità e dello scopo a cui i dati inviati sono destinati, mentre l'ente proprietario del gruppo può eseguire analisi singole o aggregate sugli utenti partecipanti.

Modality

Codifica un'interazione body-cloud o viewer-cloud. Le modality realizzano l'idea base della progettazione, rappresentano infatti un workflow distribuito. Le modality sono pensate per estendere le capacità di un client mobile. Un client può infatti scaricare dinamicamente la lista delle modality ed eseguirle in un approccio app oriented tipico dei sistemi operativi mobile moderni.

Il *cloud* permette di accedere e manipolare le sue astrazioni tramite una REST WEB API (ovvero un web service), lasciando piena libertà su quali servizi e quali tecnologie utilizzare per la realizzazione delle applicazioni client. Il web service è realizzato utilizzando il framework Restlet e utilizza OAuth per autenticare le richieste. Le comunicazioni con gli altri componenti sono quindi realizzate utilizzando HTTP come protocollo e XML come formato per le rappresentazioni, eccetto per i dataset che vengono trasferiti in CSV.

Internamente le entità sono organizzate in un Domain Model[22], la persistenza degli oggetti avviene tramite il persistence layer. Il processo di data analysis viene invece eseguito in un core logico, il *workflow engine*. Il web service comunica col workflow engine mediante interfaccia, in modo da mantenere l'implementazione di questo indipendente. L'interfaccia del workflow engine è compatibile con un generico pipelining tool, in modo da renderne possibile l'implementazione mediante l'utilizzo di uno strumento professionale di terze parti. I processi decisionali sono definiti da workflow compatibili con l'implementazione dell'engine.

Il cloud si occupa solamente di storage e analisi dei dati, mentre la visualizzazione dell'output è affidata completamente al client, che può tuttavia attenersi alle specifiche definite nelle view.

L'intero SaaS si affida a Google App Engine (GAE) che fornisce un Servlet Container (ambiente di esecuzione per web application JavaEE) e vari servizi cloud based. La Datastore API offerta da GAE permette di accedere al suo datastore, ed è utilizzata per implementare il persistence layer; mentre la Task Queue API è utilizzata dal web service (attraverso un'interfaccia Java) per mettere in coda le richieste.

3.5 Progettazione del cloud-side

3.5.1 Casi d'uso

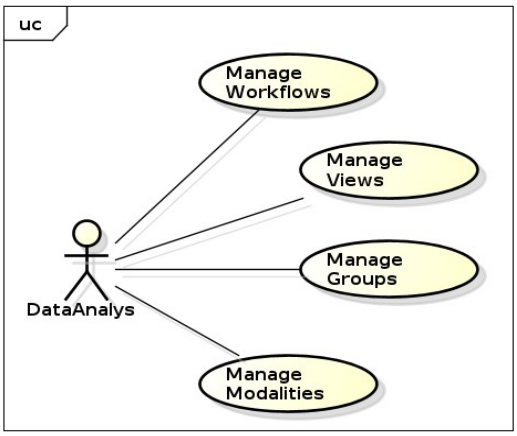

Figure 16: Data Analyst use case

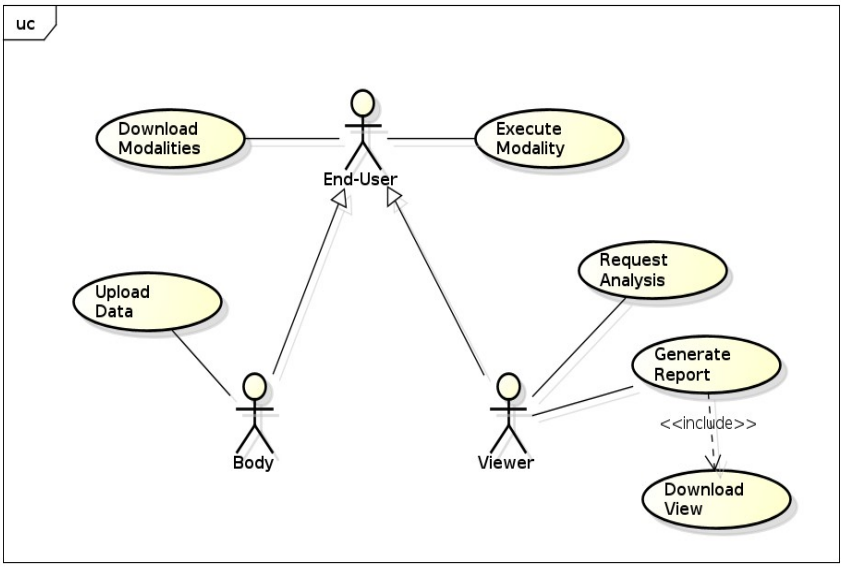

Figure 17: End-User use case

- Il data analyst configura la piattaforma a seconda delle esigenze degli utenti finali, definendo le entità principali del sistema. L'analyst può quindi eseguire operazioni CRUD (create, read, update, delete) su workflow, group, view e modality.

- L'utente finale può accedere alle operazioni supportate dal cloud attraverso le modality, oppure può utilizzare direttamente le funzioni di basso livello, a seconda della tipologia di client utilizzato. È possibile distinguere due tipologie di utenti: body, colui che invia al cloud i dati provenienti dalla BSN; e viewer, colui che richiede l'analisi dei dati associati a uno o più utenti, e che può riceverne l'output, eventualmente generato utilizzando una view.

3.5.2 Domain Model

Il Domain Model, ovvero le classi che modellano il dominio applicativo sono le seguenti:

`Entity.`

Classe astratta rappresentante un'entità del sistema che può essere identificata e distinta dalle altre. Una Entity ha un tipo, rappresentato dalla sua classe concreta; un nome, stringa che la identifica univocamente tra le altre entità dello stesso tipo; e UUID (universal unique identifier), stringa alfanumerica di lunghezza fissa che la identifica univocamente tra tutte le entità.

`User`

Un utente del sistema. Il server non prevede una diretta distinzione dei ruoli degli utenti, un utente può virtualmente compiere qualunque operazione.

`Describable`

Classe astratta che aggiunge alcune proprietà ad Entity. `Describable` è un'entità a cui è associato un utente proprietario, l'unico che può modificarla o eliminarla, e che può essere descritta attraverso meta-dati.

`Metadata`

Semplice classe contenente meta-dati a scopo descrittivo.

`Group`

Identifica un particolare servizio di analisi e soprattutto funge da aggregatore di dati destinati al servizio stesso. Un gruppo è caratterizzato da tre attributi: specifiche di input, ovvero le specifiche dei dati accettati dal gruppo; members, gli utenti autorizzati ad eseguire analisi sui dati associati al gruppo; enrolled, attributo opzionale che restringe la possibilità di inviare dati ai soli utenti specificati.

`Workflow`

Identifica una sequenza di operazioni da compiere sui dati. La logica di analisi è implementata da un altro componente, perciò la classe Workflow è semplicemente un contenitore di dati binari da dare in input al Workflow Engine.

`View`

Contiene le istruzioni per generare la rappresentazione grafica di un dataset.

`DataTable`

Contenitore di dati provenienti da una BSN. È associata ad un utente proprietario e al gruppo a cui i dati sono destinati. Un domain model coerente richiederebbe includere i dati in un'istanza di DataTable, anch'essi con una rappresentazione ad oggetti, ma data la loro natura dinamica e l'importanza che ricoprono la loro persistenza è affidata direttamente al persistence layer.

`DataSpecification`

Codifica la specifica di dati in formato tabulare. Contiene a sua volta le specifiche di ciascuna colonna di dati ed eventualmente l'URI di una View. Ogni colonna contiene un nome indicativo, un tipo primitivo (integer, double, long, string), ed eventualmente un sorgente che rappresenta il sensore da cui i dati provengono.

`ServerAction`

Identifica una chiamata alla WEB API del cloud, specificando URI e metodo HTTP da utilizzare.

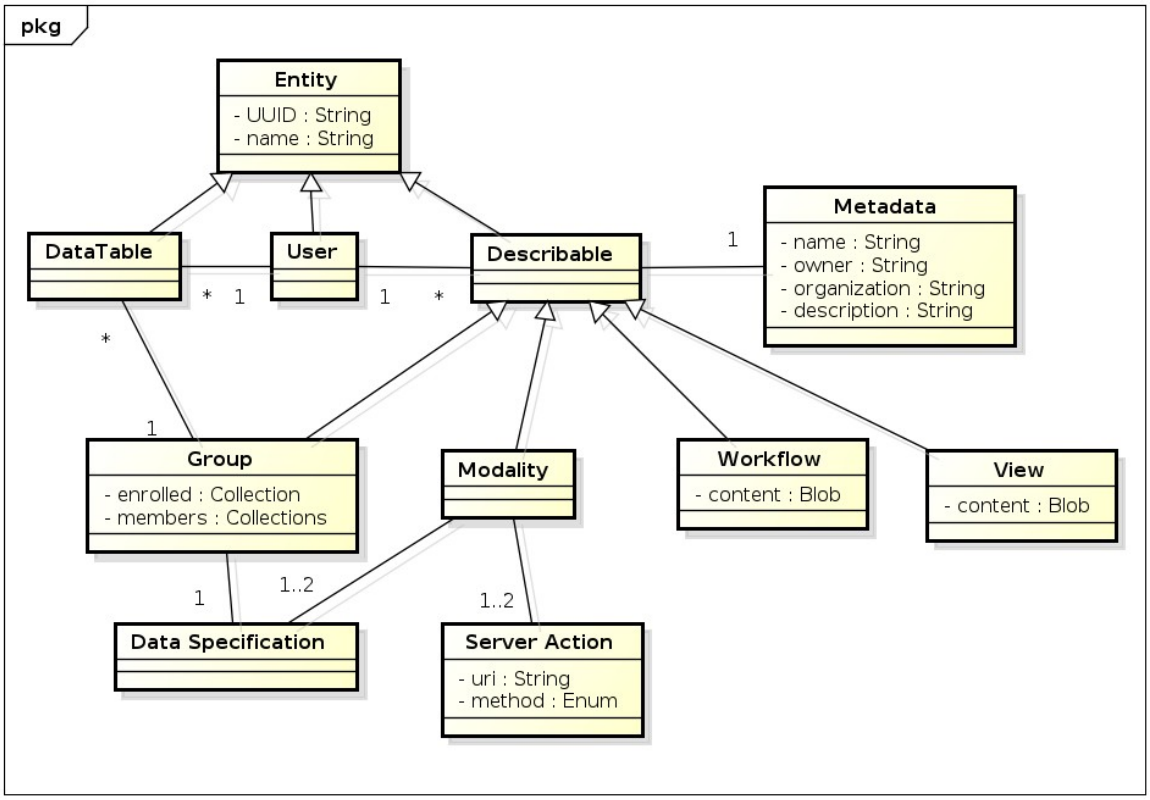

Figure 18: Domain Model

`Modality`

Rappresenta un'operazione che il client può effettuare. Contiene le specifiche dei dati in input, utilizzabile da client per determinare quali sensori della BSN utilizzare e il formato dei dati da inviare al cloud; una `ServerAction` di inizializzazione del cloud (opzionale); la `ServerAction` che identifica l'operazione principale che il cloud deve eseguire e le specifiche dei dati in output che questa può restituire.

3.5.3 Rappresentazione dei dati

La rappresentazione di dati generici riveste un ruolo fondamentale nel sistema. I dati sono organizzati in insiemi (dataset) con formato tabulare. Ogni colonna del dataset rappresenta una particolare variabile. Ogni riga corrisponde a un determinato membro o istanza del dataset. Ogni istanza è una tupla composta dai valori delle variabili.

L'intestazione della tabella, ovvero l'insieme di nomi e tipi di dato delle varie colonne

viene specificato dalla classe `DataSpecification`. Quest'ultima include inoltre due caratteristiche inerenti il dominio applicativo: `source` e `view`. `source` può essere specificato per una singola colonna ed indica il sensore da utilizzare per generare la sequenza di dati corrispondente, mentre view può essere usato per specificare l'URI della specifica da utilizzare per la visualizzazione dei dati. La DataSpecification può essere definita in XML seguendo la struttura mostrata nello snippet 3.

```
<dataSpecification>
  <column>
    <name/>
    <type/>
    <source/>
  </column>
  <view/>
</dataSpecification>
```

Snippet 3: DataSpecification XML structure

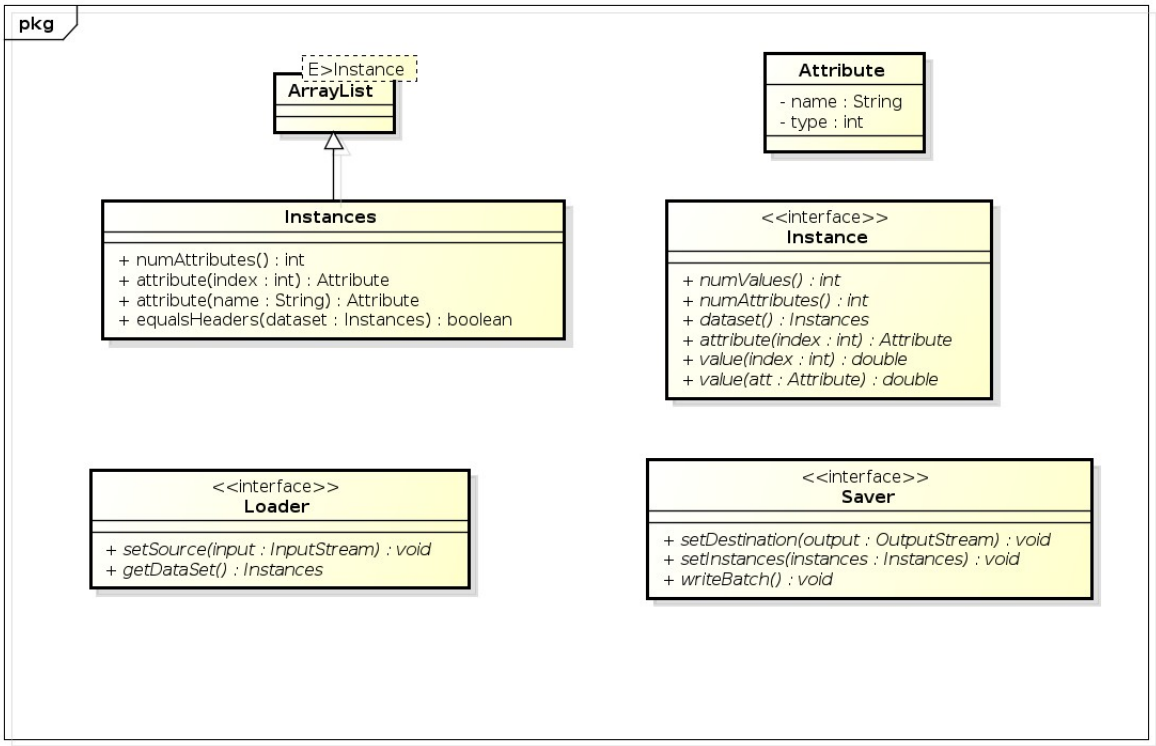

Figure 19: The subset of the weka API used in this project

Data la grande quantità di software open source esistenti in questo campo, si è scelto di

utilizzare uno di questi per la gestione interna dei dataset. La scelta è ricaduta sulle librerie di weka[10] in quanto complete, efficienti, e senza particolari dipendenze.

Le colonne vengono definite dalla classe Attribute, l'interfaccia Instance rappresenta una singola riga. La classe Instances contiene un insieme ordinato di attributi, ovvero l'intestazione del dataset, e un insieme ordinato di Instance.

Loader e Saver forniscono due interfacce per convertire i dati da e verso vari formati, tra cui CSV, Arff e Json.

3.5.4 Persistence Layer API

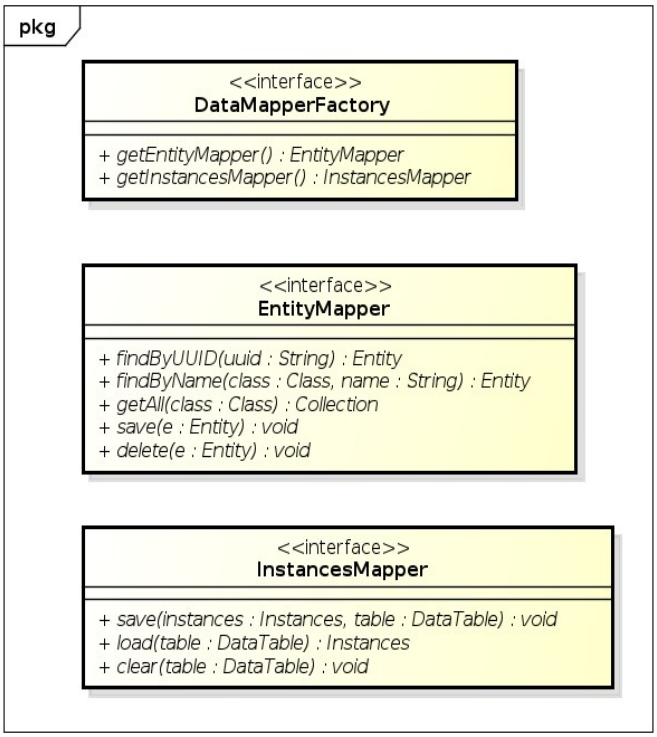

Figure 20: Persistence Layer API

L'API del Persistence Layer è basata sul Data Mapper[22] pattern, ovvero su classi adibite al mapping degli oggetti sul data source.

Le istanze del Domain Model vengono salvate mediante l'interfaccia `EntityMapper`, che fornisce inoltre metodi di ricerca basati su nome, tipo e UUID.

10 http://www.cs.waikato.ac.nz/ml/weka/

I dataset, contenuti in istanze della classe `Instances`, vengono salvate tramite l'interfaccia `InstancesMapper`. L'`InstancesMapper` definisce tre metodi:

`save`
Inserisce le istanze nella `DataTable` specificata.

`load`
Carica tutte le istanze da una data `DataTable`.

`clear`
Cancella tutte le istanze associate ad una `DataTable`.

3.5.5 Workflow Engine API

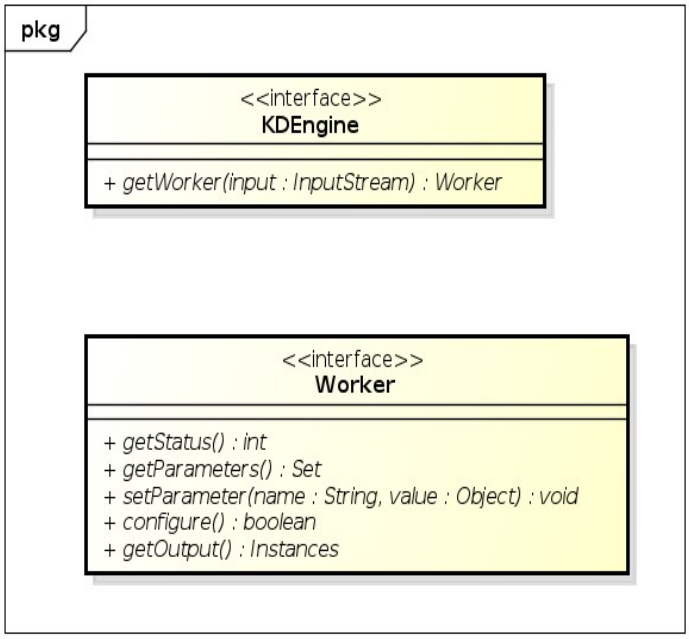

Figure 21: Workflow Engine API

L'API del Workflow Engine è progettata per permettere di utilizzare come implementazione software di terze parti.

L'interfaccia `KDEngine` ha un unico metodo, il quale prende come parametro un workflow, sotto forma di stream, ed istanza un `Worker`.

L'interfaccia `Worker` si occupa di configurare e controllare l'esecuzione di un'operazione di analisi. Estende l'interfaccia Java `Runnable` per permettere di eseguire l'operazione in un thread separato.

Il `Worker` non permette di definire alcun input, il workflow stesso deve essere configurato per leggere i dati necessari all'analisi. Ciò perfette di definire opportune query direttamente nel workflow e rende possibile leggere dati anche da sorgenti esterne.

Un'istanza di `Worker` ha un ciclo vita che può essere interrogato tramite il metodo `getStatus`. Gli stati possibili che un `Worker` può assumere sono elencati di seguito.

Waiting Configuration

Lo stato iniziale. Il `Worker` aspetta che gli venga passato un eventuale set di parametri, definito in base all'implementazione della classe e al workflow stesso, tramite il metodo `setParameter`; e che venga successivamente chiamato il metodo `configure()`.

Error Wrong Config

Non è stato possibile configurare l'operazione. La causa può essere un parametro errato o mancante.

Ready

La configurazione è terminata correttamente e l'operazione può essere eseguita.

Error Wrong Input

C'è stato un errore nella lettura dell'input durante l'esecuzione del workflow.

Error Runtime

L'operazione è terminata a causa di un errore non meglio identificato.

Job Completed

L'operazione è terminata correttamente.

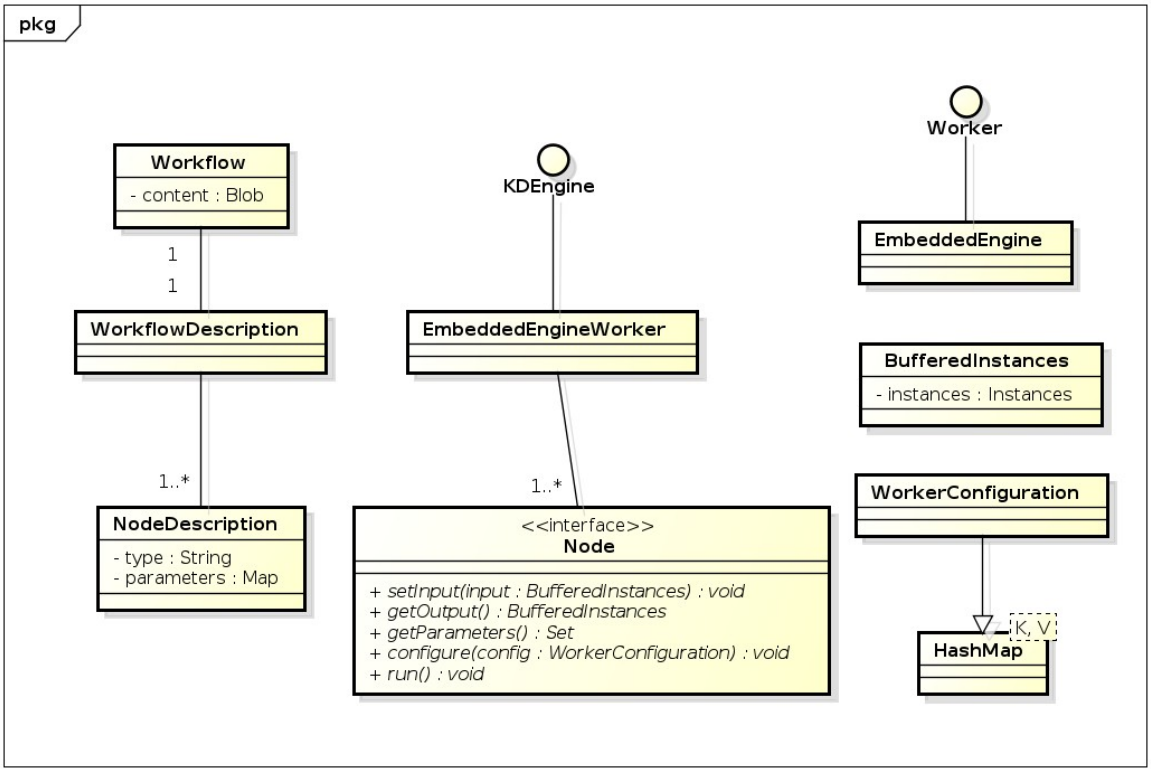

Figure 22: Embedded Engine Architecture

Embedded Engine

Ai fini di test e del lavoro di tesi è stata realizzata anche una semplice implementazione del Workflow Engine integrata nel server.

L'Embdedded Engine supporta workflow composti da una singola sequenza di nodi. Ogni nodo prende in input un dataset dal nodo precedente (eccetto il primo che dovrebbe leggere i dati da un *data source*), e produce un dataset in output per il nodo successivo, l'output dell'ultimo nodo corrisponde all'output finale del workflow.

La classe `WorkflowDescription` rappresenta un workflow e contiene le informazioni necessarie all'engine per istanziare una sequenza di `Node`. Un workflow può essere definito dagli utenti in XML e passato all'engine come stream. L'engine ricava il corrispondente oggetto Java tramite un'operazione di unmarshalling, utilizzando la libreria JAXB. Il documento XML rappresentante il workflow è composto da una sequenza ti elementi `node`, ognuno dei quali include un elemento `type`, ovvero il nome di una classe Java che implementa l'interfaccia `Node`, ed eventualmente una serie di coppie nome-valore che identificano parametri con cui istanziare la suddetta classe.

1. **Configurazione**. Un Node può richiedere parametri a runtime (**parametri dinamici**), specificando i nomi tramite il metodo `getParameters`. L'omonimo metodo di `EmbeddedEngineWorker` restituisce i nomi di tutti i parametri richiesti dai nodi del workflow, e inserisce tutti i parametri ricevuti in `WorkerConfiguration`. La fase di configurazione viene eseguita dal Worker richiamando il metodo `Node.configure` e permette al nodo di recuperare i parametri richiesti dalla `WorkerConfiguration`. Tali parametri sono dinamici, uno stesso workflow può quindi essere eseguito più volte con valori diversi.

```
<workflow>
  <node>
    <type/>
    <parameter>
      <name/>
      <value/>
    </parameter>
  </node>
</workflow>
```

Snippet 4: Workflow Description structure

`Node` è l'interfaccia per le operazioni atomiche dell'engine, siano esse operazioni di lettura dati, filtri o algoritmi di elaborazione. Così come `Worker`, anche `Node` ha un suo ciclo vita descritto dalle seguenti fasi:

2. **Istanziazione**. Un Node viene istanziato in base al suo tipo, ovvero il nome della corrispondente classe Java, specificato in `NodeDescription`. L'oggetto può anche avere attributi di classe i cui valori saranno settati subito dopo l'istanziazione, in base a un set di parametri definiti nella description (**parametri statici** inclusi nel workflow).
3. **Input/Output**. Il nodo riceve un riferimento al dataset in input, che può non essere completamente caricato in memoria, ma contiene le informazioni necessarie per la validazione. Allo stesso modo un nodo fornisce un riferimento al suo output al successivo, mediante l'oggetto contenitore `BufferedInstances`.
4. **Elaborazione**. Il nodo esegue l'elaborazione vera e propria dei dati, mediante il metodo `run()`.

3.5.6 Web Service front-end

Il cloud espone le funzionalità attraverso un web service, il quale:

- Offre al mondo esterno un'astrazione del Domain Model, basato su rappresentazioni in XML, permettendo di effettuare le operazioni CRUD sulle entità.
- Offre una API per le funzionalità del Workflow Engine, con caratteristiche specifiche in base all'implementazione utilizzata.
- Accede agli elementi del back-end tramite interfacce, è quindi facilmente portabile e platform-independent.
- Permette ai client di mettere eventualmente in coda le richieste, evitando così di mantenere connessioni aperte per lunghe operazioni di elaborazione.
- È facilmente estendibile, permettendo ai programmatori di implementare nuovi servizi con minimo sforzo.
- È realizzato utilizzando il framework Restlet.

Architettura

Di seguito vengono definiti i componenti del web service, la cui architettura è illustrata in figura 23.

`KDServerResource`

Classe astratta, una generica risorsa del web service. Estende la classe `ServerResource`, un'istanza rappresenta lo stato attuale di una risorsa e viene creata per gestire una singola richiesta. Incorpora metodi di utilità e riferimenti ai componenti di back-end, i quali vengono iniettati dall'application context. È la classe base di tutte le risorse.

`TaskQueue`

Interfaccia che permette di mettere in coda una richiesta, in modo da essere eseguita asincronicamente in un secondo momento.

`KDApplication`

Processa le richieste alle risorse. Ogni nuova risorsa deve essere registrata tramite questa classe.

`MainApplication`

L'unica classe platform-aware del server. Si occupa di inizializzare l'application context e di inserirvi le istanze di TaskQueue, KDEngine e DataMapperFactory. Inoltre pre-processa le richieste dirette a KDApplication controllando l'autenticazione (affidata all'Authenticator), e la confidenzialità (tutte le richieste HTTP sono re-direzionate in HTTPS dal Redirector).

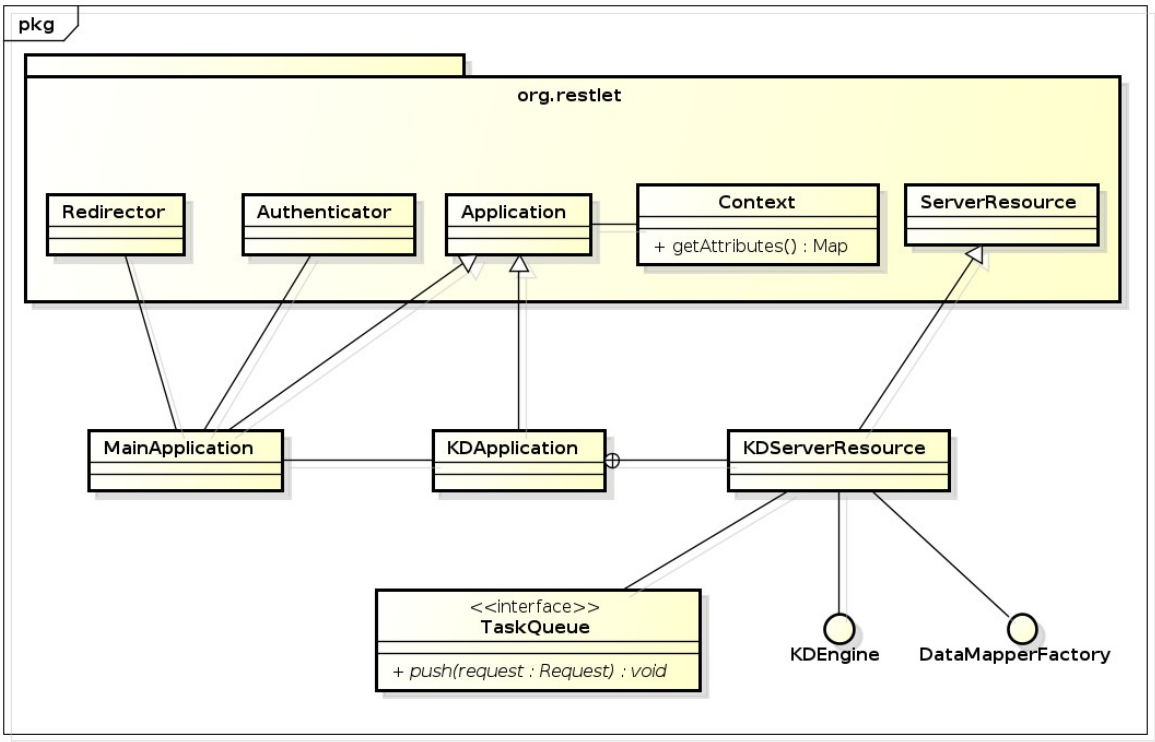

Figure 23: Web Service Architecture

Risorse

Il web service definisce cinque tipologie di risorse:

- Ciascuna classe del Domain Model che estende `Describable` è una risorsa. È identificata da un path del tipo `/<NomeClasse>`, supporta il solo metodo `GET`, e

restituisce gli URI di tutte le istanze di quel tipo. Un utente può decidere di filtrare solo le istanze di cui è proprietario aggiungendo all'URI la query `filter=owned`.

- Ciascuna istanza della classe `Describable` è una risorsa. È identificata da un path del tipo `/<NomeClasse>/<NomeIstanza>`, in cui `NomeClasse` identifica la classe concreta di appartenenza dell'istanza, e `NomeIstanza` il nome dell'oggetto, univoco per la classe. I metodi supportati da questo tipo di risorsa sono `PUT`, `GET`, `DELETE`.
- Risorse che dipendono da un particolare gruppo, il cui path è concatenato a quello di un'istanza di `Group`. Tali risorse sono:

  `/data`
  Rappresenta i dati dell'utente che effettua la richiesta nel gruppo indicato. Supporta i metodi PUT, GET, DELETE.

  `/contributors`
  La lista degli utenti partecipanti, che hanno inviato dati al gruppo. Supporta il solo metodo GET.

  `/members`
  I membri del gruppo, ovvero coloro che possono accedere ai dati di tutti i partecipanti. Supporta il solo metodo GET.

  `/enrolled`
  La lista degli utenti autorizzati ad inviare i propri dati al gruppo. Supporta il solo metodo GET.

  Members ed enrolled possono essere modificate dal proprietario del gruppo col metodo POST.
- La risorsa metadata, identificata da un path del tipo `/metadata/<UUID>`, è associata ai meta-dati dell'oggetto col dato UUID. Supporta i metodi PUT e GET.
- Le risorse che dipendono dal Workflow Engine, il cui path è concatenato al prefisso `/engine`

Rappresentazioni

Il formato utilizzato per le rappresentazioni è XML. Le rappresentazioni vengono generate automaticamente da Restlet a partire da oggetti Java, grazie all'estensione di Restlet per JAXB. L'unica eccezione risiede nella rappresentazione dei dataset, per il quale si utilizza il formato CSV in quanto standard per la codifica di tabelle, evitando inoltre in questo modo l'overhead che una rappresentazione XML avrebbe generato con grandi quantità di dati.

Embdedded Engine WEB API

Il web service espone un sottoinsieme di API adatte al particolare engine che si utilizza. Le risorse definite per interagire con l'embedded engine sono due:

`/workflow/<name>`

Un workflow identificato univocamente da `<name>`. Un workflow può essere creato e modificato dal suo proprietario col metodo PUT, e letto da chiunque col metodo GET. Il metodo POST invece richiama l'embedded engine per l'esecuzione del processo di analisi. La richiesta deve contenere una web form con tutti i parametri di runtime necessari all'esecuzione del workflow. Al termine dell'elaborazione dell'engine, l'output, in formato CSV, viene inserito nella risposta HTTP. Siccome alcune elaborazioni potrebbero durare a lungo, è anche possibile utilizzare la TaskQueue in modo da non lasciare il client in attesa, aggiungendo all'URI la query `queue=yes`.

`/node/<name>`

Un nodo utilizzabile nella composizione di workflow. La rappresentazione della risorsa è un jar contenente una classe che implementa l'interfaccia `Node`, e tutte le altre classi da cui questa dipende. `<name>` è il nome della classe ed identifica il nodo nella composizione dei workflow.

3.6 Modality

Le modality rappresentano un workflow distribuito. L'idea alla base è di codificare una generica interazione client-server, in modo da fornire ai client una sorta di applicazione bodycloud confezionata, rendendo possibile agli utenti utilizzare diverse modalità del server e di scaricarne di nuove non appena vengono definite.

Sono state prese in considerazione tre diverse tecniche per giungere a questo obiettivo:

1. Codificare le modality in Java, rendendone possibile l'utilizzo sui client come plugin. Sebbene offri massima flessibilità, ciò avrebbe reso necessario realizzare le applicazioni client in Java, utilizzando inoltre un'opportuna libreria per il caricamento dei plugin. Come se non bastasse il client dovrebbe ottenere dal server bytecode Java, ma il bytecode della Dalvik Virtual Machine utilizzata su Android è diverso dal quello di una normale JVM.
2. Codificare le modality in python, linguaggio di scripting general purpose, soluzione meno flessibile della precedente ma che non pone vincoli sul client in quanto l'interprete python è disponibile su ogni piattaforma.. Nel caso peggiore questa soluzione avrebbe richiesto di includere l'interprete nel client, appesantendo l'applicazione. Inoltre un client dovrebbe essere fornito di un insieme di librerie e strutture dati per compiere le azioni più comuni.
3. Codificare le modality in un linguaggio ad-hoc di facile interpretazione. Sebbene le operazioni eseguibili dal client siano limitate dai costrutti definiti dal linguaggio, lo sforzo implementativo di questa soluzione è minimo e non pone alcun vincolo su client.

La scelta è ricaduta quindi sulla terza opzione, utilizzando XML per esprimere i semplici costrutti necessari, in modo da facilitare il parsing e mantenersi in linea col formato utilizzato dal web service.

Una modality è rappresenta quindi da un documento XML diviso in quattro sezioni (come mostrato nello snippet 5):

1. **Formato dei dati in input**. Opzionale. Se presente istruisce il client su quali sensori utilizzare per raccogliere i dati e il formato degli stessi. La specifica è composta da colonne, ciascuna delle quali ha un nome, un tipo (integer, double, timestamp, ecc.), e il sensore sorgente dei valori da collezionare.
2. **Chiamata di inizializzazione al web service**. Opzionale. Contiene il path di una

risorsa e un metodo HTTP, indica una richiesta da effettuare al server, per inizializzarlo o per recuperare parametri necessari all'esecuzione vera e propria. La chiamata non dovrebbe mai essere una `PUT`, perché non vi è modo di definire una risorsa da inviare al server. Nel caso del metodo `POST`, è necessario definire un set di coppie nome-valore, da inviare nel body della richiesta, codificati come post form.

3. **Chiamata di esecuzione al web service**. In modo analogo alla chiamata di inizializzazione, indica una richiesta da effettuare al server, ma può includere due ulteriori costrutti (snippet 6):
    - *Repeat*. Può essere true o false. Se true la chiamata dev'essere ripetuta fino a che l'utente non interrompe l'esecuzione della modality.
    - *Trigger*. Identifica una condizione di attivazione, la richiesta al server può essere verificata solo quando questa si verifica. Attualmente l'unico trigger supportato è basato su un timer, ma la specifica può essere facilmente estesa.

Inoltre se il metodo è `PUT` il client deve inviare i dati raccolti. Nel caso del metodo `POST` invece, oltre a specificare coppie nome-valore, è possibile specificare coppie nome-riferimento. Il riferimento è a sua volta composto da un'espressione XPath e da un attributo `type`. In base al valore di quest'ultimo vi sono due scenari possibili:

`CHOICE`

Il client valuta l'espressione XPath sull'output della chiamata di inizializzazione, successivamente deve richiedere l'intervento dell'utente per selezionare quale dei risultati dell'espressione bisogna utilizzare come valore del parametro indicato.

`MAP`

Il client valuta l'espressione XPath sull'output della chiamata di inizializzazione, successivamente esegue una richiesta al server per ognuno dei valori restituito dall'espressione. Il termine *map* fa riferimento alla programmazione funzionale. Nella programmazione funzionale una *map* è una funzione che applica un elemento di una data funzione ad una lista di elementi e restituisce in output una lista di risultati.

4. **Formato dei dati in output**. Opzionale. Se presente, indica al client che la chiamata di esecuzione restituisce un dataset conforme a questo formato. La specifica è composta da colonne così come nella fase 1 (con la differenza che la direttiva `source` non avrebbe senso), ma a differenza delle precedente può contenere l'URI di una view, che il client può richiedere al server, tramite una richiesta HTTP `GET`, per generare un report. Il report può essere generato utilizzando la libreria jxReport che, integrando dati e view, genera codice HTML, facilmente visualizzabile dal client con un normale browser o HTML viewer.

Il diagramma di attività in figura 24 mostra le operazioni che un client dovrebbe svolgere interpretando una modality.

```
<modality>
 <inputSpecification/>
 <init-action/>
 <action/>
 <outputSpecification/>
</modality>
```

Snippet 5: Modality XML Structure

```
<action>
 <uri/>
 <method/>
 <parameter>
   <name/>
   <value/>
   <reference xpath="" type=""/>
 <parameter/>
 <repeat/>
 <trigger/>
</action>
```

Snippet 6: Action XML structure

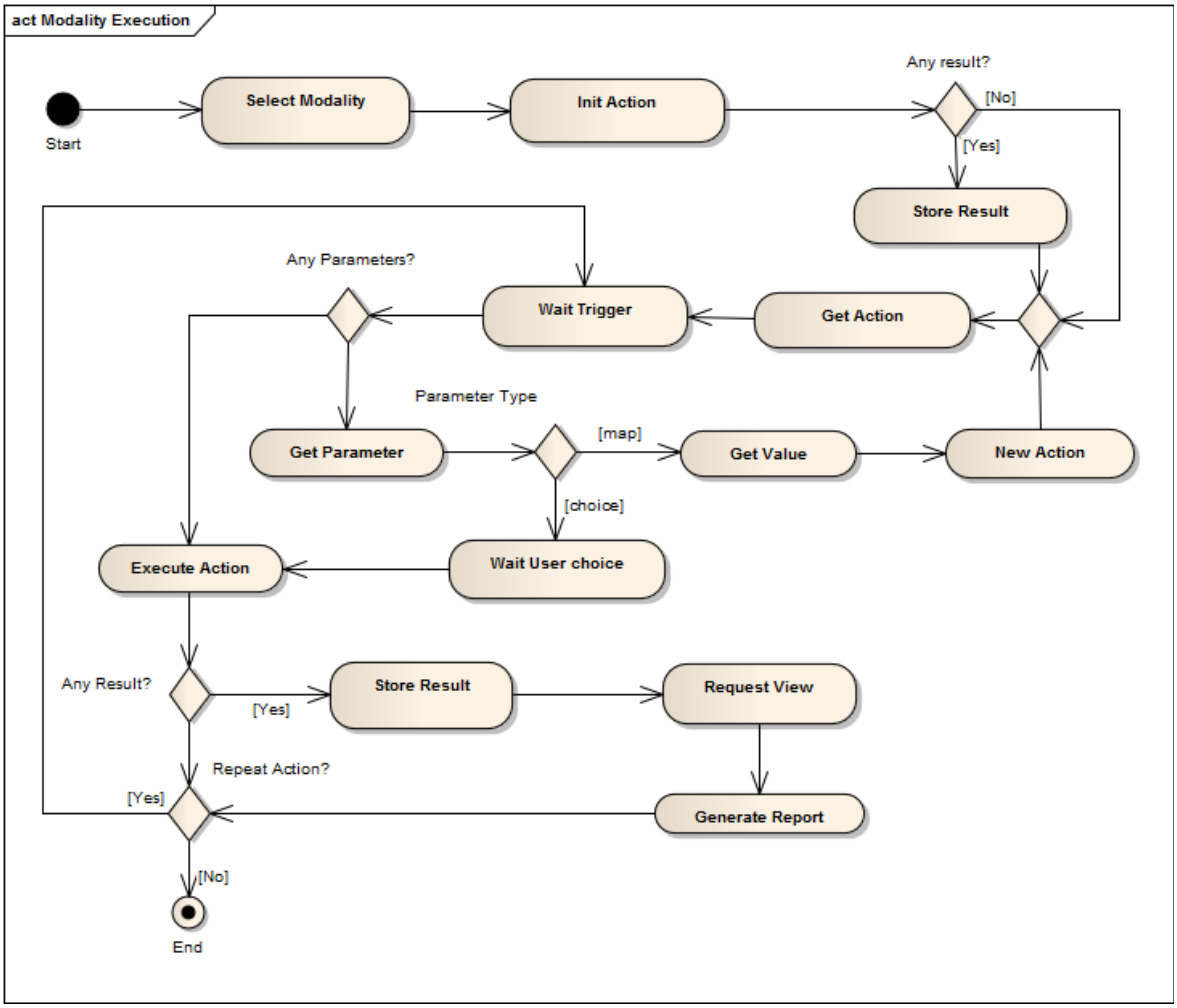

Figure 24: Modality execution activity

3.7 Sviluppo di nuovi servizi

Il data analyst può creare nuovi servizi definendo workflow (e nodi), gruppi, view e modality. Ogni entità può essere creata tramite una richiesta HTTP `PUT` alla corrispondente risorsa del web service. La semplicità di tale operazione non richiede nient'altro che un client HTTP come applicazione lato data analyst.

Un servizio completo è incentrato sul processo di analisi da eseguire. La definizione del servizio può essere schematizzata nei seguenti punti:

1. **Algoritmi**. Implementazione e caricamento degli algoritmi necessari, come nodi del Workflow Engine. Tutti i nodi caricati vengono memorizzati e messi a disposizione di tutti gli altri utenti.
2. **Data Source**. Definizione di un gruppo contenente le specifiche dei dati che possono essere raccolti dalla BSN ed elaborati dagli algoritmi.
3. **Workflow**. Definizione del processo di analisi vero e proprio, tramite la combinazione dei nodi esistenti, configurati opportunamente. Il primo nodo dovrebbe leggere i dati dal Data Source.
4. **View**. Definizione di una o più rappresentazioni grafiche dell'output del workflow.
5. **Modality**. Dovrebbero essere definite almeno una modality per il body e una per il viewer. La modality destinata al body dovrebbe avere un input analogo alla specifica dei dati del Data Source, una action adibita al caricamento dei dati nel gruppo definito nella fase 2 e nessuna specifica di output. La modality destinata al viewer dovrebbe lanciare l'esecuzione del workflow definito al punto 3 come action, i parametri di questa devono essere definiti in base ai parametri dinamici richiesti dai vari nodi. Le specifiche dell'output della modality devono essere conforme all'output del workflow e contenere il riferimento ad una delle view definite nel punto 4.

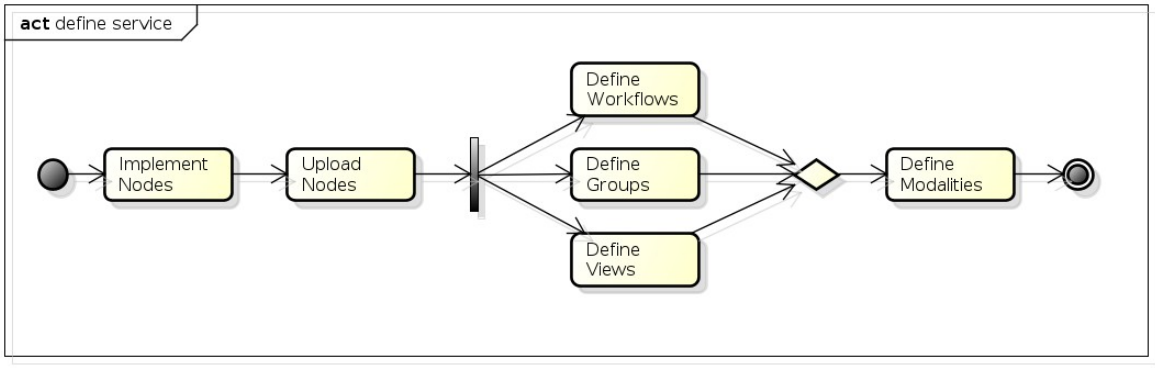

Figure 25: Service definition activity diagram

3.8 Comunicazione Client-Server

I diagrammi successivi illustrano una generica interazione client-server per ciascuno degli attori coinvolti.

Analyst-Cloud

1. Definizione di un workflow.
2. Definizione di un gruppo.
3. Definizione di una view.
4. Definizione di una modality.

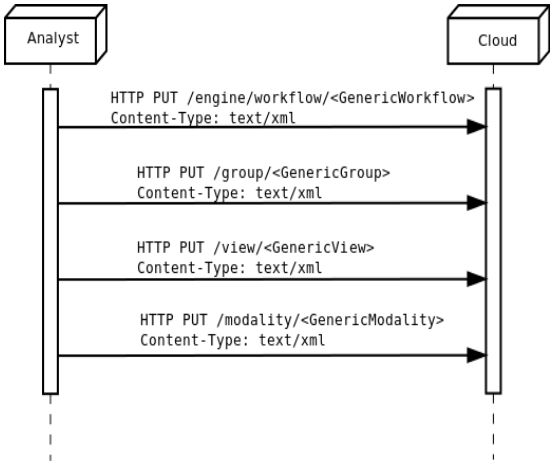

Figure 26: Analyst-Cloud interaction

Body-Cloud

1. Richiesta della lista di tutte le modality disponibili. Il cloud risponde con un documento XML contenente gli URI delle modality e dei relativi meta-dati.
2. Richiesta delle specifiche della modality che si intende eseguire, utilizzando uno degli URI ricevuti in precedenza.
3. I dati generati dalla BSN vengono formattati in CSV e inviati al cloud, tramite una richiesta HTTP, ad intervalli regolari.

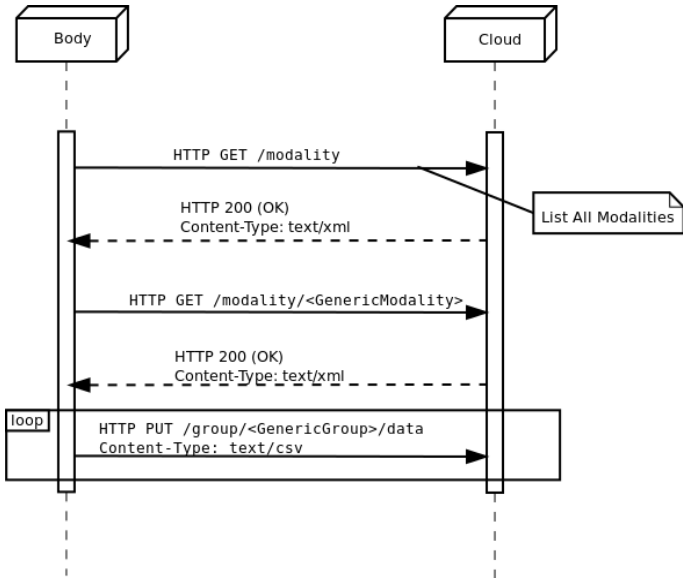

Figure 27: Body-Cloud interaction

Viewer-Cloud

1. Richiesta della lista di tutte le modality disponibili. Il cloud risponde con un documento XML contenente gli URI delle modality e dei relativi meta-dati.
2. Richiesta delle specifiche della modality che intende eseguire, utilizzando uno degli URI ricevuti in precedenza.
3. Richiesta HTTP `POST` alla risorsa che identifica un workflow, così come indicato nella modality. La richiesta deve contenere tutti i parametri necessari al determinato workflow, quali utente e gruppo su cui effettuare l'analisi. Al termine dell'elaborazione, il viewer riceve i dati in output, codificati in CSV.
4. Richiesta della view, tramite HTTP `GET` all'URL indicato nella modality.

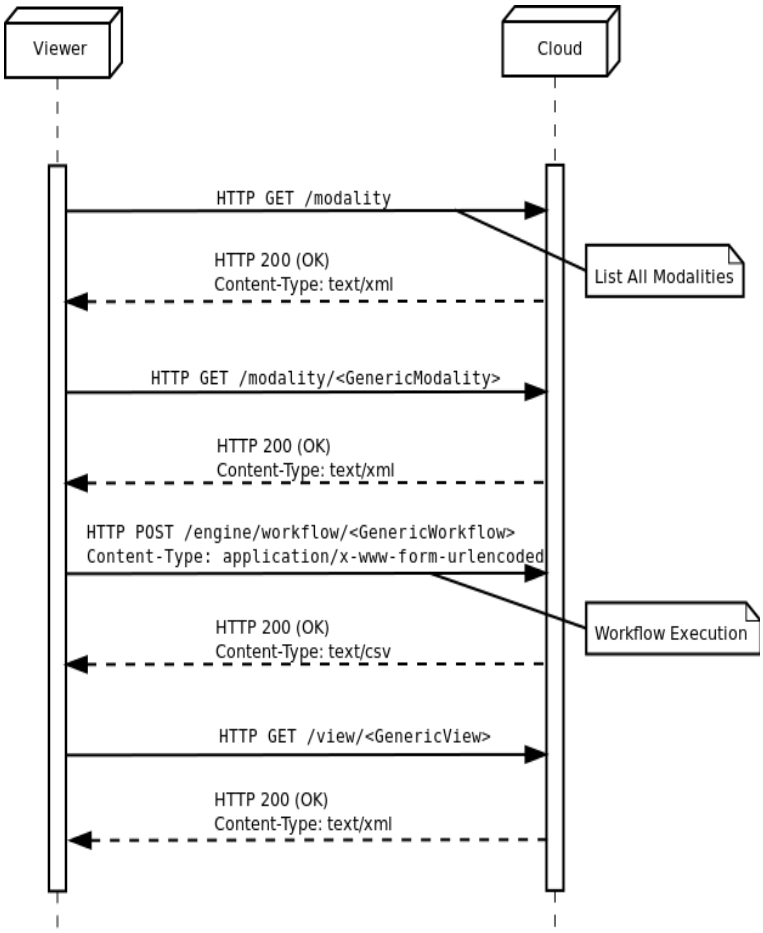

Figure 28: Viewer-Cloud interaction

4 Realizzazione e testing

4.1 Implementazione del Persistence Layer

Il Persistence Layer utilizza come data source il datastore di GAE, pertanto i mapper devono essere implementati utilizzando una delle API fornite dal PaaS.

Per l'implementazione dell'`EntityMapper` risulta più appropriato utilizzare una delle API di alto livello per la persistenza trasparente degli oggetti, vale a dire JPA o JDO. Entrambe utilizzano DataNucleus come provider di persistenza, congiuntamente ad un apposito plugin per renderlo compatibile con il datastore. La scelta è ricaduta su JDO, in quanto gode di un supporto più completo su GAE, in particolare rende possibili il mapping trasparente di tutte le relazioni tra le classi. Scendendo a basso livello nel datastore di GAE infatti è possibile distinguere tra due tipi di relazioni, quella tra entità padre ed entità figlia (*owned*), e quella tra due entità che non appartengono alla stessa gerarchia (*unowned*). Il mapping trasparente delle relazioni *unowned* è possibile solo con JDO.

Le classi la cui persistenza è gestita da JDO sono solo le discendenti di `Entity`. Il mapping è stato definito dalle annotazioni Java, per cui ogni classe persistente è annotata come `@PersistenceCapable`. Gli attributi `uuid` e `name` della classe `Entity` vengono associate rispettivamente alla *key* della corrispettiva entità del datastore (quindi generata automaticamente) e al *key name*, come mostrato nello snippet 7.

```
@PrimaryKey
@Persistent
@Extension(vendorName="datanucleus", key="gae.encoded-pk", value="true")
private String uuid;

@Persistent
@Extension(vendorName="datanucleus", key="gae.pk-name", value="true")
private String name;
```

Snippet 7: uuid and name mapping

L'ereditarietà delle classi deve essere anch'essa mappata da JDO, tramite quelle che vengono definite *inheritance strategy*. GAE ne supporta due, *subclass-table*, che inserisce gli attributi di una classe nelle tabelle relative alle classi figlie, e *complete-table*, che mappa gli attributi di una classe e di tutti i suoi ancestor in una medesima tabella. La strategia scelta è *complete-table*, e dev'essere specificata nell'entità radice della gerarchia:

```
@Inheritance(strategy=InheritanceStrategy.COMPLETE_TABLE)
```

Le relazioni sono marcate dall'annotazione `@Persistent`, le relazioni unowned sono accompagnate anche dall'omonima annotazione `@Unowned`. Le classi che non fanno parte della gerarchia di `Entity` invece non sono gestite direttamente da DataNucleus ma vengono incluse nelle tabelle relative alle classi persistenti come oggetti serializzati. Gli attributi che devono essere serializzati sono marcati dalla seguente annotazione:

```
@Persistent(serialized=”true”)
```

L'`EntityMapper` non necessita di query particolari, gli oggetti sono letti dal datastore per `uuid`, nome o tipo; tutte modalità supportate direttamente dal `PersistenceManager` di JDO. È importante notare che ogni classe persistente ha un proprietario (il proprietario di un oggetto `User` è l'utente stesso), ciò riduce al minimo la possibilità di scritture concorrenti di una stessa entità, operazione che GAE non supporta e che si ripercuote su una delle due transazioni con un'eccezione (`ConcurrentModificationException`, attualmente non gestita dal web service).

L'implementazione di `InstancesMapper` invece, prevedendo tabelle dinamiche utilizza direttamente le API di basso livello del datastore. Dato un dataset, ciascuna riga

viene mappata in un'entità come *child* di un'istanza di `DataTable`, facendo corrispondere ad ogni elemento della tupla una *property*. L'operazione di lettura di un dataset avviene invece effettuando una ancestor query specificando la chiave della `DataTable` di appartenenza. Anche in questo caso non vi sono grossi problemi di concorrenza, dato che ogni utente inserisce i dati nella propria `DataTable`, il mapper gestisce tuttavia la `ConcurrentModificationException` nelle operazioni di inserimento. In questo caso infatti risulta molto semplice risolvere il conflitto: la transazione interrotta viene ripetuta dopo un breve lasso di tempo e accoda i dati a quelli inseriti in precedenza.

4.2 Integrazione dell'Embedded Engine

Il Workflow Engine deve poter accedere al data source, per questo sono stati sviluppati tre diversi nodi, i quali utilizzano le funzioni del persistence layer, e richiedono come parametri di runtime i relativi oggetti mapper:

- `UserDataReader`. Legge i dati associati ad un singolo utente all'interno di un gruppo. Oltre ai mapper richiede come parametri dinamici nome utente (`sourceUser`) e gruppo (`sourceGroup`) da cui recuperare i dati.
- `GroupDataReader`. Legge tutti i dati associati ad un gruppo. Oltre ai mapper richiede come parametro dinamico il gruppo (`sourceGroup`) da cui recuperare i dati.
- `UserDataWriter`. Scrive i dati su un determinato gruppo, associandoli ad un particolare utente. Oltre ai mapper richiede come parametri dinamici nome utente (`destinationUser`) e gruppo (`destinationGroup`) di destinazione.

4.3 Integrazione del Web Service

GAE per Java è simile ad un Servlet Container JavaEE, è sufficiente quindi configurare il file web.xml affinché tutti gli URL vengano mappati nella ServerServlet fornita da Restlet, come mostrato nello snippet 8.

```
<servlet>
      <servlet-name>RestletServlet</servlet-name>
      <servlet-class>org.restlet.ext.servlet.ServerServlet</servlet-class>
      [...]
</servlet>
<servlet-mapping>
      <servlet-name>RestletServlet</servlet-name>
      <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Snippet 8: ServerServlet mapping in web.xml

4.4 Autenticazione

Il sistema non integra un proprio sistema di gestione degli account, ma permette a chiunque possieda un account Google di autenticarsi tramite OAuth. Nei dettagli l'applicazione client di un utente deve essere in possesso di un access token, da utilizzare per autenticare ciascuna richiesta.

L'access token può essere recuperato in modi diversi, a seconda della piattaforma utilizzata dal client. La procedura generale[11] richiede che il client sia preventivamente registrato nella Google API Console. L'endpoint dell'Authentication Server di Google a cui inviare la richiesta del token è il seguente:

```
https://accounts.google.com/o/oauth2/auth
```

La richiesta prevede inoltre un set di parametri nella query string, riportati in tabella 4:

11 https://developers.google.com/accounts/docs/OAuth2Login

Table 4: the set of query string parameters to use against the Google Authentication Server

Parameter	Values	Description
`response_type`	`token`	Determines thet the Google Authorization Server must return an access token
`client_id`	the client_id obtained from the APIs Console	Indicates the client that is making the request. The value passed in this parameter must exactly match the value shown in the APIs Console
`redirect_uri`	one of your `redirect_uris` registered at the APIs Console	Determines where the response is sent. The value of this parameter must exactly match one of the values registered in the APIs Console
`scope`	space delimited set of permissions the application requests	Indicates the Google API access your application is requesting. The values passed in this parameter inform the consent page shown to the user. There is an inverse relationship between the number of permissions requested and the likelihood of obtaining user consent.

Table 5: Scopes for User Profile Permission

Scope Value	Description
`https://www.googleapis.com/auth/userinfo.profile`	Gain read-only access to basic profile information, including a user identifier, name, profile photo, profile URL, country, language, timezone, and birthdate.
`https://www.googleapis.com/auth/userinfo.email`	Gain read-only access to the user's email address.

Una volta in possesso del token, il web service deve poter recuperare le informazioni relative all'accout utente, perciò è necessario specificare opportunamente gli scope, mostrati nella tabella 15:

Il token deve poi essere incluso in ogni richiesta al web service, utilizzando l'header HTTP Authorization e lo schema Bearer[12]:

```
Authorization: Bearer {token}
```

Quando il web service riceve una generica richiesta, richiede a sua volta la validazione del token all'Authorization Server, tramite il seguente URI:

```
https://www.googleapis.com/oauth2/v1/tokeninfo?access_token={accessToken}
```

Se il token è valido il web service riceverà in risposta le informazioni relative all'utente, altrimenti esso riceverà un messaggio di errore e considererà la richiesta non autenticata.

Tale sistema di autenticazione, oltre a non richiedere un particolare sforzo

12 http://tools.ietf.org/html/rfc6750

implementativo, risulta particolarmente adatto per l'autenticazione di client Android (ogni dispositivo Android è in genere associato ad un account Google).

4.5 Setup del progetto

La gestione del processo di sviluppo del software è affidato a Maven, il quale offre un notevole aiuto riguardo la gestione del gran numero di dipendenze richieste, automatizzazione degli unit test e deploy su GAE. Il software è stato diviso in quattro artifact (ovvero quattro moduli):

`weka-stripped`
Contiene il subset delle API di Weka utilizzate per la rappresentazione dei dati.

`bodycloud-lib`
Un set di classi Java utilizzate dal web service ma distribuite separatamente per permetterne l'inclusione in applicazioni client.

`bodycloud-engine`
Embedded Engine.

`bodycloud-server`
Il server cloud vero e proprio.

4.5.1 weka-stripped

Il modulo è stato estratto direttamente dal codice sorgente di weka. Contiene i seguenti package:

`weka.core`
Tutte le classi precedentemente descritte (`Instance`, `Instances`, `Attribute`) e loro dipendenze.

`weka.core.converters`

Contiene le classi adibite al caricamento e al salvataggio dei dati nei vari formati (CSV, Arff, Json).

4.5.2 bodycloud-lib

Il modulo contiene un set di librerie utili per la realizzazione di un client Java. Come tale l'artifact ha tra le sue dipendenze alcuni moduli di Restlet. Di seguito la descrizione dei package.

`org.bodycloud.lib.bean`

I Java bean che vengono automaticamente serializzati da Restlet in XML durante la comunicazione.

`org.bodycloud.lib.rest.api`

Le interfacce Java delle risorse Restlet.

`org.bodycloud.lib.rest.ext`

Risorse personalizzate (estensioni della classe `org.restlet.resource.Resource`), in particolare la classe `InstancesRepresentation`, utilizzata per il trasferimento dei dataset.

`org.bodycloud.lib.client`

Implementazione astratta di un client interprete di modality. Può essere utilizzata come punto di partenza per la realizzazione di un'applicazione client vera e propria.

4.5.3 bodycloud-engine

L'engine, che dipende direttamente da weka-stripped è distribuito nei seguenti package:

`org.bodycloyd.engine`

L'interfaccia con cui accedere all'engine e che può essere implementata da qualsiasi altro tool.

`org.bodycloyd.engine.embedded`
L'implementazione dell'embedded engine.

`org.bodycloyd.engine.embedded.helper`
classi di utilità.

`org.bodycloyd.engine.embedded.model`
i bean tramite cui è possibile definire un workflow.

`org.bodycloyd.engine.embedded.node`
il package in cui devono essere definiti i nodi. I nodi forniti direttamente da questo modulo sono `FileDataReader`, capace di leggere un dataset da file, e `ConsoleOutput`, che stampa il suo input su console.

4.5.4 Bodycloud-server

Il presente artifact contiene web service, domain model e persistence layer. Le dipendenze da gestire sono molteplici e includono le API J2EE, GAE, JDO e Restlet. Di seguito vengono elencati in dettaglio gli artifact specificati nel pom:

groupId: org.bodycloud
artifactId: bodycloud-lib
Il modulo bodycloud-lib

groupId org.bodycloud
artifactId bodycloud-engine
Il modulo bodycloud-engine

groupId com.google.appengine
artifactId appengine-api-1.0-sdk
API di Google App Engine.

groupId javax.jdo
artifactId: jdo-api

API di JDO.

groupId org.datanucleus
artifactId: datanucleus-core
Implementazione di JDO.

groupId com.google.appengine.orm
artifactId: datanucleus-appengine
Plug-in di GAE che permette di utilizzare DataNucleus come persistence provider per il datastore.

groupId org.restlet.gae
artifactId: org.restlet.ext.servlet
Librerie necessarie per il deploy del web service Restlet nel Servlet Container J2EE di GAE.

groupId org.restlet.gae
artifactId: org.restlet.ext.xml
Libreria per la conversione e gestione di rappresentazioni XML.

groupId org.restlet.gae
artifactId: org.restlet.ext.jaxb
Libreria necessaria a runtime per effettuare in modo trasparente la conversione dei Java bean in XML e vice versa.

groupId org.restlet.gae
artifactId: org.restlet.ext.net
Client connector per Restlet.

I package che compongono il modulo sono:

`org.bodycloud.server.rest`
web service.

`org.bodycloud.server.domain`

domain model.

`org.bodycloud.server.persistence`

Interfaccia del layer di persistenza.

`org.bodycloud.server.persistence.gae`

Implementazione del layer di persistenza adatta per GAE.

`org.bodycloyd.engine.embedded.node`

I nodi dell'embedded engine che fanno uso del persistence layer, ovvero `UserDataReader`, `UserDataWriter`, e `GroupDataReader`.

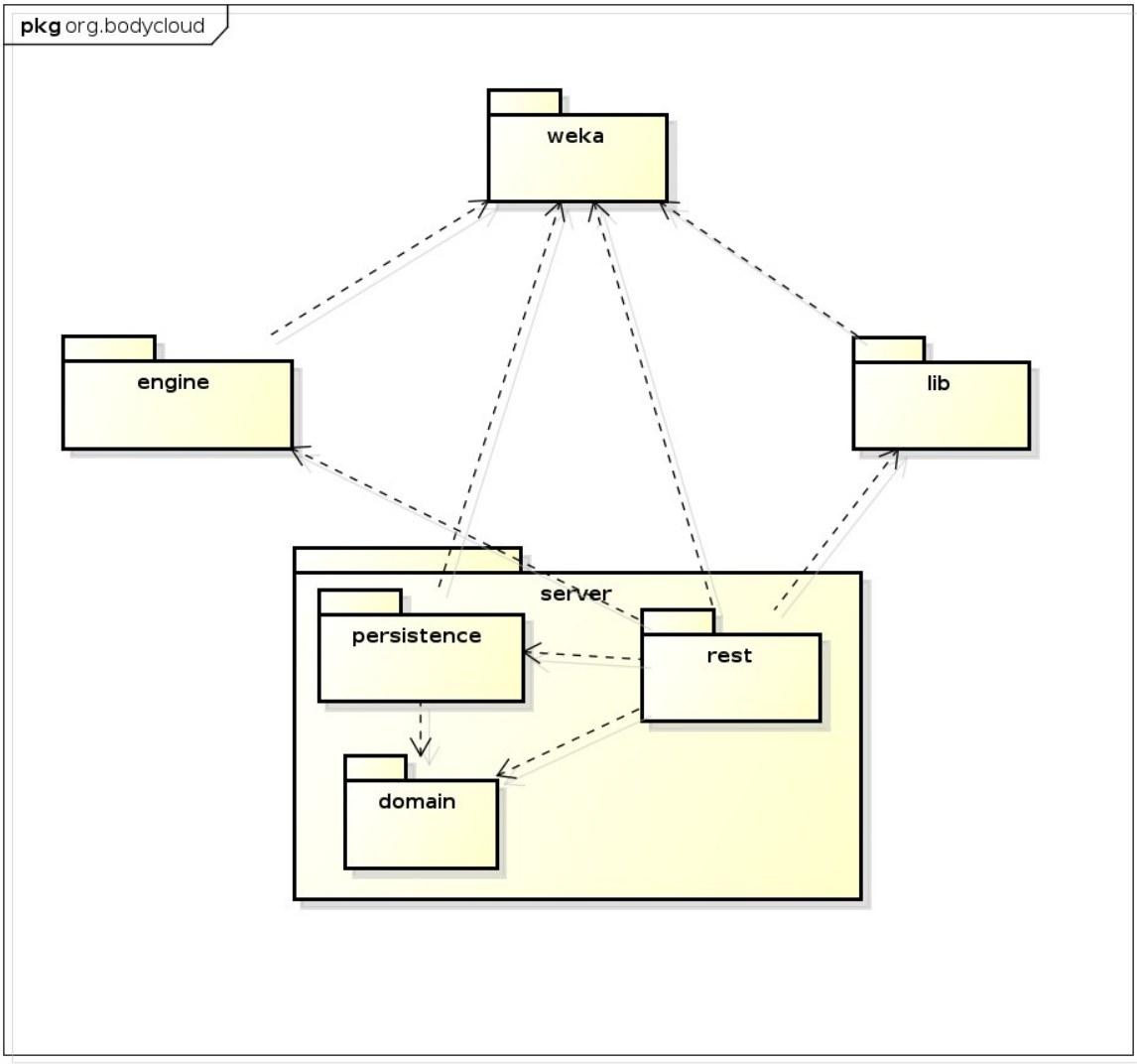

Figure 29: Packages

4.5.5 Sviluppo di nuovi nodi

Per facilitare lo sviluppo di nuovi nodi dell'embedded engine è stato realizzato un apposito artifact, `embedded-engine-node-archetype`. L'artifact contiene un nodo template che può essere usato come punto di partenza, e un semplice ambiente di test utile a verificare il corretto funzionamento del plugin. Lo sviluppatore non deve fare altro che sviluppare il nodo (o i nodi), avendo cura di mantenerli nello stesso package (`org.bodycloyd.engine.embedded.node`). I nodi sviluppati possono comunque utilizzare altre classi e l'artifact risultante può contenere altri package. Quello che un nodo non dovrebbe fare invece è accedere alle funzioni del sistema operativo (socket e filesystem) in quanto tali operazioni non sono permesse nel runtime environment di GAE.

Una volta implementato il proprio codice non bisogna far altro che inserire il nodo risultante nel workflow `src/test/resources/test-case.xml`. Il workflow di test utilizza il nodo `FileDataReader`, che può essere configurato nel documento XML per utilizzare un particolare file, per caricare il dataset da dare in input al nodo da testare. L'output può eventualmente essere visualizzato durante il test aggiungendo al workflow il nodo `ConsoleOutput`.

A questo punto Maven si occuperà di eseguire il test e di creare l'opportuno archivio jar da caricare sul cloud, il tutto con il comando:

```
mvn package
```

4.6 Caso d'uso: monitor dell'ECG

Per la fase di sperimentazione è stato realizzato un sistema per il monitoraggio di dati provenienti da un sensore cardiaco. L'applicazione fa riferimento ad una necessità reale, si immagini un gruppo di atleti impegnati in una competizione sportiva, l'applicazione permetterebbe ad un gruppo di medici di monitorare le loro condizioni cardiache in modo da individuare eventuali anomalie preventivamente.

Il lato body prevede quindi un utente provvisto di sensore cardiaco e client Android, il viewer può essere l'utente stesso oppure un monitore autorizzato.

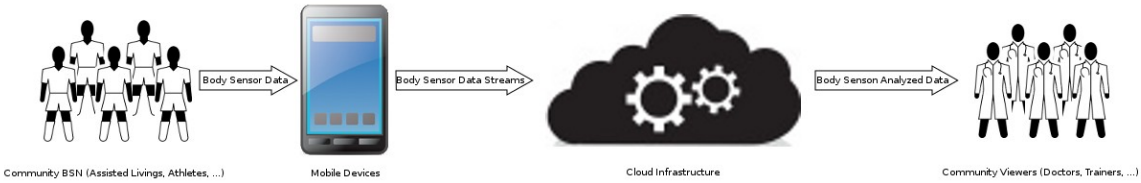

Figure 30: Bodycloud ECG monitoring

4.6.1 Workflow

Il segnale ECG digitalizzato è rappresentabile come un vettore numerico, i cui valori sono campionati ad una frequenza regolare. L'algoritmo in questione riceve in input un vettore e calcola il cosiddetto segnale R-R, ovvero una sequenza di tempi di inter-battito (RR) espressi in ms. Il workflow è composto da due nodi: lo `UserDataReader` legge l'input dal datastore, `RR` implementa l'algoritmo sopracitato. Il nodo `RR` può essere riutilizzato in altri workflow, dal segnale R-R è ad esempio possibile calcolare la frequenza cardiaca, ma la sperimentazione non si è spinta oltre.

```
<workflow>
  <node>
    <type>UserDataReader</type>
  </node>
  <node>
    <type>RR</type>
  </node>
</workflow>
```

Snippet 9: Workflow for R-R signal calculation

4.6.2 Formato dei dati

Il dataset corrispondente al vettore ECG è composto da una sola colonna, generata dal sensore denominato `ECGShimmerSensor`. Di seguito l'XML corrisponde alla specifica.

```
<dataSpecification>
  <data>
    <name>ECGShimmerSample</name>
    <type>INTEGER</type>
    <source>ECGShimmerSensor</source>
  </data>
<dataSpecification>
```

Snippet 10: ECG dataset specification

4.6.3 View

```
<report>
  <lineChart>
    <title>Tachogram</title>
    <showLegend>false</showLegend>
    <showMarker>false</showMarker>
    <line>
      <lineLabel>First Line</lineLabel>
      <lineColor>blue</lineColor>
      <lineWidth>1</lineWidth>
      <showLine>true</showLine>
      <points>
        <xMarker>
          <counter/>
        </xMarker>
        <yMarker>
          <index>0<index/>
        </yMarker>
      </points>
    </line>
  </lineChart>
</report>
```

Snippet 11: Tachogram view specification

A partire dall'output dell'algoritmo, ovvero il segnale R-R, è possibile visualizzare il cosiddetto tacogramma, ovvero un diagramma a linee composto dal segnale sull'asse delle ordinate ed un numero progressivo sull'asse delle ascisse.

4.6.4 Modality

Sono state definite tre modality, una per l'utilizzo lato body, le altre due per l'utilizzo lato viewer. Tutte fanno riferimento ad un unico gruppo, ecg-monitoring.

Data Feed

```
<modality>
  <inputSpecification>
    <data>
      <name>ECGShimmerSample</name>
      <type>INTEGER</type>
      <source>ECGShimmerSensor</source>
    </data>
  </inputSpecification>
  <init-action>
    <uri>/group/ecg-monitoring/data</uri>
      <method>DELETE</method>
  </init-action>
  <action>
    <uri>/group/ecg-monitoring/data</uri>
      <method>PUT</method>
      <repeat>true</repeat>
      <trigger after="60"/>
  </action>
</modality>
```

Snippet 12: Data Feed

La modality data feed è pensata esclusivamente per inviare un feed di dati proveniente dal sensore cardiaco al cloud. Le specifiche di input sono conformi al formato dei dati definito in precedenza. La action viene ripetuta ogni minuto ed invia i dati al gruppo ecg-monitoring. I dati raccolti dalla modality devono essere attinenti ad una stessa prestazione sportiva (o altro evento), per questo la init-action si occupa di eliminare eventuali dati raccolti precedentemente.

Single Analysis

```
<modality>
  <init-action>
    <uri>/group/ecg-monitoring/contributors</uri>
    <method>GET</method>
  </init-action>
  <action>
    <uri>/engine/workflow/ecg</uri>
    <method>POST</method>
    <parameter>
      <name>sourceUser</name>
      <reference xpath="//users/user"/>
    </parameter>
    <parameter>
      <name>sourceGroup</name>
      <value>ecg-monitoring</value>
    </parameter>
    <repeat>false</repeat>
  </action>
  <outputSpecification>
    <data>
      <name>rr</name>
      <type>DOUBLE</type>
    </data>
    <view>/view/tachogram.xml</view>
  </outputSpecification>
</modality>
```

Snippet 13: Single ECG Analysis

La single analysis è utilizzabile per ottenere il tacogramma di un determinato utente monitorato. La init-action richiede al cloud la lista degli utenti che hanno inviato dati al gruppo ecg-monitoring. La lista completa viene restituita solo agli utenti membri del gruppo, in questo caso si suppone siano medici autorizzati. Una volta ricevuta la lista, il client del viewer dovrebbe richiedere all'utente di selezionare l'atleta che si vuole monitorare. Effettuata la scelta, il client richiederà l'analisi dell'ECG associato all'atleta selezionato. Ricevuto l'output è possibile scaricare la view e visualizzare il tacogramma con jxReport.

La single analysis può anche essere utilizzata dall'atleta stesso, la init-action in questo caso infatti restituirà una lista col solo identificatore dell'utente richiedente.

Group Analysis

La group analysis è del tutto analoga alla precedente, ma a differenza della single analysis, istruisce il client a eseguire una richiesta di elaborazione per ogni utente elencato nella lista dei partecipanti. Il comportamento della modality cambia rispetto alla precedente semplicemente specificando l'attributo `type=MAP` nella reference del parametro riferito all'utente da monitorare.

```
<modality>
  <init-action>
    <uri>/group/ecg-monitoring/contributors</uri>
    <method>GET</method>
  </init-action>
  <action>
    <uri>/engine/workflow/ecg</uri>
    <method>POST</method>
    <parameter>
      <name>sourceUser</name>
      <reference xpath="//users/user"/ type=”MAP”>
    </parameter>
    <parameter>
      <name>sourceGroup</name>
      <value>ecg-monitoring</value>
    </parameter>
    <repeat>false</repeat>
  </action>
  <outputSpecification>
    <data>
      <name>rr</name>
      <type>DOUBLE</type>
    </data>
    <view>/view/tachogram.xml</view>
  </outputSpecification>
</modality>
```

Snippet 14: Group ECG Analysis

4.6.5 Testing e analisi delle prestazioni

Il caso d'uso è stato testato eseguendo un data feed simultaneamente su una serie di client, simulati sulle macchine a disposizione in laboratorio. Il sensore cardiaco genera approssimativamente 6000 campioni al minuto, pertanto ciascun client simulato è stato configurato per effettuare 10 richieste, a intervalli di 60 secondi, ciascuna delle quali invia al cloud un dataset di 6000 tuple.

In modo da simulare un relativamente alto numero di client col ridotto numero di macchine disponibili, è stato necessario sviluppare un'applicazione ad-hoc che facesse uso di multithreading, in modo da parallelizzare il più possibile l'invio delle richieste, L'applicazione è stata realizzata in python, utilizzando la libreria requests[13] per effettuare richieste HTTP, l'applicazione può essere invocata da riga di comando specificando diversi parametri come numero di client da simulare, numero di thread da utilizzare, numero di richieste da effettuare, intervalli tra le richieste e dati da inviare; e restituisce in output una serie di statistiche sui tempi di risposta (tempo minimo, massimo, medio e deviazione standard).

Sono stati effettuati cinque diversi test, variando il numero di client simulati, da 10 a 50 con passo 10, utilizzando 10 diverse macchine fisiche (3 desktop, 5 notebook, 3 netbook). Le macchine sono state programmate per eseguire un singolo batch script ad un orario specificato, il quale richiama a sua volta l'applicazione di benchmark con diversi parametri.

I tempi di risposta possono essere osservati nel grafico sottostante. Come si evince la figura le prestazioni si mantengono grossomodo costanti così come ci si aspetta da un servizio cloud, eccetto per il picco con 50 client che però può essere giustificato da diversi fattori:

- Effetto collo di bottiglia dovuto ad una significativa quantità di traffico generata da una stessa rete.
- Utilizzo di un singolo utente, cosa che procura un'elevata concorrenza nelle operazioni di scrittura nel datastore, problema che non si pone con la multi-utenza in quanto ogni utente ha una tabella riservata su cui scrivere.
- Anomalie dovute alla gestione di un elevato numero di thread da parte di processori poco performanti (es. Intel Atom).

13 ***python-requests***.org

- Errori lato cloud dovuti all'esaurimento dello spazio disponibile nel datastore.

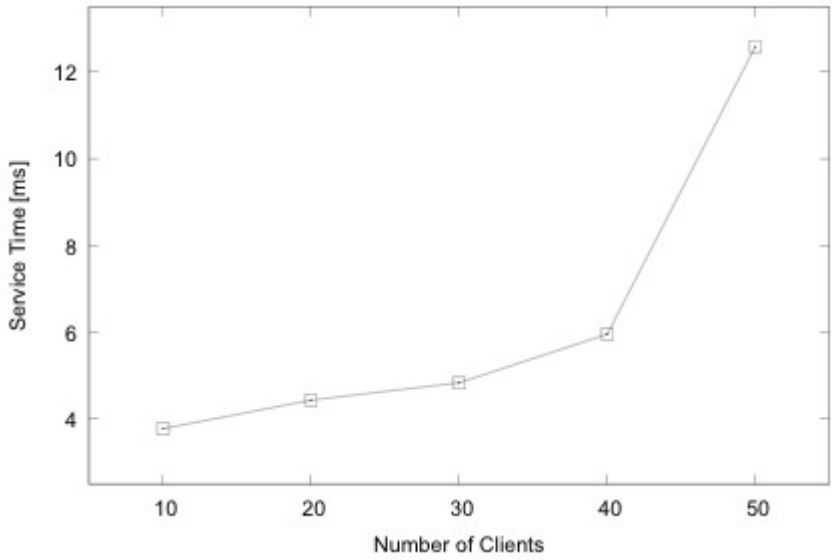

Figure 31: Scalability evaluation: DataFeed service time vs. number of clients

Conclusioni

È stato realizzato un SaaS per il BodyCloud che permette ad analisti ed esperti di dominio di fornire una serie di servizi basati sull'analisi dei dati provenienti da Body Sensor Network, e agli utenti di usufruire di tali applicazioni tramite l'uso di un dispositivo mobile; il tutto grazie ad una RESTful WEB API e una serie di entità definibili in XML.

Il server, tramite l'utilizzo dinamico di workflow, è in grado di combinare diversi algoritmi di elaborazione, mentre tramite modality e view è in grado di specificare al client che tipo di sensori utilizzare, quale workflow richiedere per una data operazione e come visualizzare il report finale del processo.

In sintesi sono stati raggiunti gli obiettivi principali della tesi, vale a dire fornire un framework BodyCloud generico ed estendibile basato su standard riconosciuti.

Tra gli sviluppi futuri vi è sicuramente l'integrazione con un tool di analisi professionale, in modo da avere già a disposizione una vasta gamma di algoritmi da utilizzare; nonché l'ottimizzazione delle operazioni di lettura/scrittura dei dati dallo storage, aspetto che può diventare determinante nell'elaborazione di dataset di grandi dimensioni.

In secondo luogo è possibile migliorare l'API, in particolare aggiungendo nuovi costrutti alle modality, in modo da avere un maggior controllo sulle operazioni eseguite dal client; oppure fornire un SDK ai data analyst in modo da evitare di dover editare manualmente diversi file XML.

Altra caratteristica importante riguarderebbe la comunicazione tra gli utenti, aspetto che aprirebbe le porte a una vasta serie di applicazioni quali riabilitazione o attività sportive assistite. Un medico potrebbe ad esempio assistere a distanza un paziente in fisioterapia, comunicando allo stesso gli esercizi da svolgere e allo stesso tempo analizzandone i risultati; o allo stesso modo un allenatore potrebbe seguire e correggere l'allenamento di un atleta.

Indice delle figure

Simple KNIME Workflow....5
Convergence of various advances leading to the advent of cloudcomputing....7
Cloud Computing Service Models....9
AppScale architecture....11
Overall Restlet Design....32
Restlet Component....32
Restlet Application Layers....33
Decomposition ofan abstract resource into Restlet artifacts....34
Subclasses of Representation....35
Character-basedrepresentation classes....35
Router distpatching calls....37
Unmarshalling example....39
Protocol Flow....47
Maven artifact directory structure....49
Bodycloud Architecture....55
Data Analyst use case....58
End-User use case....58
Domain Model....61
The subset of the weka API used in this project....62
Persistence Layer API....63
Workflow Engine API....64
Embedded Engine Architecture....66
Web Service Architecture....69
Modality execution activity....75
Service definition activity diagram....76
Analyst-Cloud interaction....77
Body-Cloud interaction....78
Viewer-Cloud interaction....79
Packages....89
Bodycloud ECG monitoring....91
Scalability evaluation: DataFeed service time vs. number of clients....97

Indice dei Code Snippet

REST XML Representation Example....19
jxReport XML example....43
DataSpecification XML structure....62
Workflow Description structure....67
Modality XML Structure....74
Action XML structure....74
uuid and name mapping....81
ServerServlet mapping in web.xml....83
Workflow for R-R signal calculation....91
ECG dataset specification....92
Tachogram view specification....92
Data Feed....93
Single ECG Analysis....94
Group ECG Analysis....95

Indice delle tabelle

JPA and JDO API overview....31
Data type bindings....38
Available axes....40
the set of query string parameters to use against the Google Authentication Server....84
Scopes for User Profile Permission....84

Bibliografia

1: C. Sieb *, T. Meinl, M. R. Berthold. Parallel And Distributed Data Pipelining With Knime, University of Konstanz, 2007

2: Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kotter, Thorsten Meinl, Peter Ohl, Kilian Thiel and Bernd Wiswedel. KNIME – The Konstanz Information Miner, University of Konstanz, 2006

3: Peter Mell, Tim Grance. The NIST Definition of Cloud Computing, National Institute of Standards and Technology, 2011

4: Eric Schmidt. Search Engine Strategies Conference, 2006, http://www.google.com/press/podium/ses2006.html

5: R. Buyya, J. Broberg, A.Goscinski. Cloud Computing: Principles and Paradigms,Wiley John + Sons, 2011

6: Simson Garfinkel. Architects of the Information Society, Thirty-Five Years of the Laboratory for Computer Science at MIT,The MIT Press, 1999

7: Jon Stokes. Guide to Virtualization, 2008, http://arstechnica.com/gadgets/2008/08/virtualization-guide-1/

8: Chandra Krintz, Chris Bunch, Navraj Chohan. AppScale: Open-SourcePlatform-As-A-Service, University of California, 2011

9: Mukaddim Pathan, Giancarlo Fortino, Ziyuan Wang. BodyCloud: Integration of Cloud Computing and Body Sensor Networks, CSIRO ICT Centre, Australia - DEIS, Università della Calabria - CSIRO Marine and Atmospheric Research, Australia,

10: Roy Fielding. Architectural Styles andthe Design of Network-based Software Architectures, University of California, 2000

11: Leonard Richardson, Sam Ruby. RESTful Web Services ,O'Reilly Media, 2007

12: Alex Rodriguez. RESTful Web services: The basics, 2008, http://www.ibm.com/developerworks/webservices/library/ws-restful/

13: Andrea Chiarelli. REST e sicurezza HTTP, 2011, http://www.html.it/pag/19607/rest-e-sicurezza-http/

14: Charles Severance. Using Google App Engine,O'Reilly Media, 2009

15: Dan Sanderson. Programming Google App Engine,O'Reilly Media, 2012

16: Roland Barcia, Geoffrey Hambrick, Kyle Brown, Robert Peterson, Kulvir Singh

Bhogal. Persistence in the Enterprise: A Guide to Persistence Technologies,IMB Press, 2008

17: Jérôme Louvel, Thierry Templier, Thierry Boileau. Restlet in Action,Manning, 2012

18: Boris Kolpackov. An Introduction to XML Data Binding in C++, 2007, http://www.artima.com/cppsource/xml_data_binding.html

19: Daniele Parisi, Giancarlo Fortino. Progetto ed implementazione della Piattaforma BodyCloud per l'Integrazione di Reti di Sensori Indossabili con il Cloud Computing, Università della Calabria, 2012

20: Dick Hardt. RFC 6749 - The OAuth 2.0 Authorization Framework, 2012

21: G. Fortino, R. Giannantonio, R. Gravina, P. Kuryloski, R. Jafari. Enabling Effective Programming and Flexible Management of Efficient Body Sensor Network Applications, IEEE Transactions on Human-Machine Systems, 2013

22: Martin Fowler. Patterns of Enterprise Application Architecture,Addison Wesley, 2002