MADiMAN

Multimedia Application for Diet Management

Title	**Deliverable WP2** **D-WP2-01**
Sub-title	Risultati delle ricerche portate avanti nel WP 2, compreso le specifiche di progetto dei protocolli di comunicazione con i repository, dell'interfaccia web, della comunicazione tra i moduli.
Document ID	D-WP2-01
Release #	1.0

Responsibilities

Role	Name	Company	E-mail	Date
Prepared by	Jelle Gerbrandy	Gerbrandy SRL	jelle@gerbrandy.com	18 novembre 2014

MADiMAN project related info

Involved WPs	2
Related Documents	fabbisogni_2013_De_Michieli.xls BANCA DATI IEO.xls
Document Type	Tecnical
Document Classification	Public

Indice

1 Introduzione

Questo Work Package si occupa del *Diet Manager Service.* Il *Diet Manager Service* è dove vengono salvati tutti i dati che sono condivisi tra i vari componenti del sistema MADiMAN. Inoltre, il Service funziona da "centro di comunicazione" tra gli altri componenti.

Questo WP ha diversi obiettivi:

1. lo studio di una infrastruttura di web service per far comunicare i vari moduli del sistema MADiMAN
2. lo studio di un sistema di repository di informazioni riguardanti l'utente, la sua dieta, la ricetta.
3. lo studio di protocolli di rete per dati multimediali strutturati.

Iniziamo in capitolo 2 con una descrizione generale dell'**architettura** del intero sistema, e descriviamo le vari parti del software che fanno parte di questo work package: il *repository* dove vengono salvati tutti i dati, l'*orchestrator* che è responsabile per la comunicazione tra i vari moduli, e, finalmente, l'*interfaccia di gestione*.

In capitolo 3 definiamo un numero di **casi d'uso,** che sono degli scenari che descrivono in modo concreto e conciso come interagisce un utente con il sistema. I casi d'uso servono come base per tutta la parte seguente, dove verrà descritto in dettaglio il funzionamento dei vari moduli.

In capitolo 4 trattiamo del **repository.** Descriviamo brevemente lo stato dell'arte, facciamo un confronto tra alcune opzioni per implementare il repository, per poi passare ad una dettagliata descrizione del modello dati.

La sezione 5 ha come soggetto l'**orchestrator,** e contiene una specifica dell'API. Dopo una desrizione dello stato dell'atre passiamo ad una dettagliate descrizione delle azioni che possono fare i vari i vari altri moduli del sistema.

Con sezione 6 concludiamo con una descrizione dell'**interfaccia di gestione dati** per l'amministrazione dei dati dell'applicazione.

2 Architettura

Il *Diet Manager Service* funziona da "centro di comunicazione" per tutti i componenti del sistema MADiMAN.

Come illustrato in Figura 1: l'ecosistema Madiman, la Applicazione Mobile (studiato in WP1), il Natural Language Parser (NLP, studiato in WP3), il Reasoner (WP4) e il modulo per Natural Language Generation (NLG, anche in WP4) comunicano attraverso il Diet Manager Service.

Per esempio, un uso tipico dell'applicazione è quello di un utente che chiede un consiglio su cosa mangiare (cf. il caso d'uso 3.2). Per presentare l'utente con un giudizio su cosa mangiare, l'Applicazione Mobile manda una richiesta al Diet Manager Service. Il Service recupera il dieta dell'utente e lo storico dei pasti che l'utente ha mangiato nel passato recente, per poi consultare il *Reasoner,* che risponde con un consiglio in un formato formale. Il Diet Manager Service poi chiede al modulo di Natural Language Generation (*NLG)* di trasformare il

consiglio formale in una risposta in linguaggio naturale. Finalmente, il service poi prepara queste riposte, e le passa all'Applicazione Mobile, che presenta la risposta all'utente.

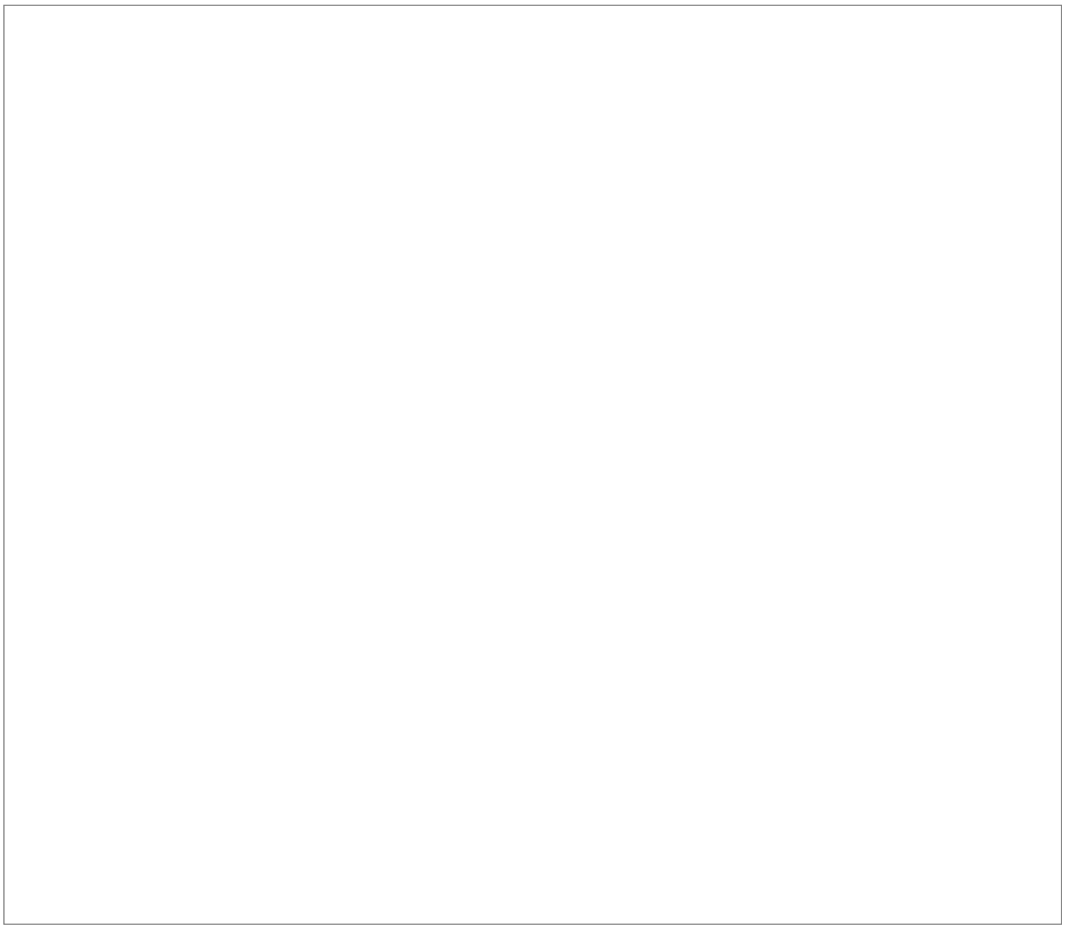


Figura 1: l'ecosistema Madiman

Il *Diet Manager Service* è diviso in tre moduli, il cui studio è l'oggetto del presente *Work Package*.

1. Il *Repository.* Qua vengono salvati i dati. Il repository contiene i dati delle ricette, le diete, le informazioni sul cibo, ma anche i dati anagrafici degli utenti, lo storico delle loro diete, ecc.
2. L'*Orchestrator* è responsabile per la comunicazione con gli altri moduli, e contiene il business logic.
3. Il *Sistema di Gestione:* un servizio sul web per amministrare i dati del repository direttamente via web.

3 Casi d'uso

Prima di procedere con lo studio dei vari moduli del Diet Manager Service, consideriamo un numero di casi d'uso che ci aiuteranno a definire le funzionalità.

Use cases - casi d'uso - sono delle descrizioni delle *funzioni* o *servizi* offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso. Sono uno strumento utile per definire dei requisiti funzionali di un sistema, e possono anche essere usati per modellare i servizi offerti da un determinato modulo o sottosistema ad altri moduli o sottosistemi.

Tipicamente, un caso d'uso descrive una concreta e specifica interazione tra un numero di "attori" - che possono essere delle persone, ma anche degli moduli software.

La formulazione di un use case ha diversi scopi: serve per definire le funzionalità di un sistema, ma serve anche come un "linguaggio comune" tra programmatori, che tendano ad usare un linguaggio molto tecnico, e clienti o utenti, che dovrebbero definire le specifiche del sistema, ma a cui talvolta mancano i concetti per esprimere in modo abbastanza preciso i requisiti del sistema.

3.1 Definizioni

Attori:

Gli "attori" sono i soggetti che interagiscono con il sistema software, o fanno parte di essa. Possono essere delle persone, ma anche componenti software. Nei casi d'uso descritti qua sotto appaiono le seguenti attori.

Persone fisiche

- Utente “normale”. Questo è l'utente del sistema che segue un dieta, chiede consigli all'applicazione, segnala quello che mangia, ecc. ecc.
- Il “dietista” è una persona di una certa competenza che può prescrive una dieta ad l’utente normale, lo consiglia nel seguire il dieta, ecc.
- “ristoratore” è il gestore di un ristorante. Gestisce delle ricette, riceve una QR-code da stampare sul menu, ecc.
- “amministratore” è responsabile per la gestione di tutti i dati nel sistema.

Servizi Software

- “app” - CheckYourMeal! mobile app
- “NLU/IE Service” - estrae dati strutturati di ricette scritte in linguaggio naturale
- “reasoner” - Reasoner Service
- “nlg” - Natural Language Generation Service
- Il "service", che è il soggetto del presente WP, che consiste di:
  - Il “repository” contiene tutti i dati, gestisce i processi ad un livello globale
  - L'Orchestrator gestisce la comunicazione tra i vari moduli
  - il "gestionale” è un interfaccia web per gestire i dati contenuti nel repository

Concetti

Nei casi d'uso qua sotto usiamo alcuni concetti ed è utile precisare quali siano.

- *Dieta*: Una dieta consiste in dei vincoli *soft* (“velocità”) di quantità di nutrienti da assumere, insieme ad una lista di cibi vietati. Per esempio, una dieta per un paziente potrebbe essere: "mangiare tra 100 e 140 grammi di proteine al giorno, ed evitare i plum cake."
- *Una Ricetta*, per il nostro sistema, *è* una tabella di ingredienti e quantità (“100 grammi di carote”) e un elenco di tecniche (esempio “frittura”) che possono influire nella dieta.
- Un *Menu* è un insieme di ricette

In capitolo 4.2 daremo delle definizioni più precisi di questi concetti.

3.2 Utente al ristorante

descrizione
Utente è al ristorante; chiede alla app se mangiare uno specifico pasto si o no

attori coinvolti
- utente
- app
- repo
- reasoner
- nlg

scenario
1. *Utente* scannerizza QR code di un piatto (o di un intero menù)
2. *App* manda ID utente + lista di piatti al *Service*
3. *Service* recupera i dati dell’utente (storico pasti, dieta) + dati dei piatti
4. *Service* chiede un’opinione al *Reasoner*
5. *Reasoner* risponde al *Service* con un giudizio motivato sulla compatibilità dei piatti con la dieta. Questo “giudizio” del reasoner si basa sulla dieta dell’utente e sui dati dei pasti che ha mangiato negli ultimi 7 giorni
6. Service chiede al module NLG di trasformare il giudizio in linguaggio naturale e/o icone (e.g. "Ieri hai già mangiato le patatine, troppi grassi saturi! ricordati di mangiare carote!")
7. *Service* manda il giudizio, insieme ai dati della ricetta, all'*App*
8. *App* visualizza la risposta al *Utente,* insieme alla domande se vuole mangiare il piatto si o no.
9. *Utente* segnale se mangiare il pasto si o no
10. *App* manda la risposta dell’utente al *Service,* che la salva

3.3 Utente mangia a casa e chiede un parere su cosa mangiare

descrizione
Questo caso d'uso è simile a il caso 3.2, con la differenza che l'utente si trova a casa, e per cui non ha un QR code a cui riferirsi

attori coinvolti
- utente
- app

- repo
- reasoner
- nlg
- nlu

scenario

1. *Utente* inserisce attraverso la app una ricetta
    a. Una ricetta nuova (forse usando una tastiera, o usando *speech recognition*) in linguaggio naturale
    b. O l'utente recupera una ricetta già inserita (in questo si saltano passi 2 e 3)
2. Il *repository* chiede al modulo *NLU* di analizzare la ricetta (N.B. si suppongono tempi di reazione *real-time*)
3. *NLU* restituisce la ricetta normalizzata (vedi definizione)
4. *Reasoner* da un giudizio sulla compatibilità del pasto con la dieta dell'utente
5. *NLG* trasforma il giudizio in una risposta in linguaggio naturale
6. *App* visualizza la ricetta normalizzata e il giudizio all'utente
7. *Utente* risponde se la ricetta è analizzato giustamente, e se mangiare il pasto si o no
8. *App* manda la risposta dell'utente al *Service,* che lo salva (data e ora incluso)

3.4 Utente registra piatti che ha già mangiato

descrizione

L'utente registra un piatto che ha mangiato in passato. La differenza con il caso sopra è che l'utente non è interessato in un giudizio sul piatto - deve solo registrare il fatto che l'ha mangiato.

attori

- utente
- app
- repo
- nlu

scenario

1. Il passo 1 è come sopra. L'app chiede anche la data e ora del pasto
2. come sopra
3. come sopra
4. *App* visualizza la ricetta normalizzata e la data, e chiede conferma all'utente
5. Il service salva i dati

3.5 Registrazione

Un utente “normale” si registra al sistema.

attori

1. utente
2. app
3. repository

scenario

1. utente scarica l’app
2. app chiede all’utente di registrarsi con i dati (vedi il modello dati per un elenco completo)
    - username
    - password
    - email
    - nome
    - cognome
    - data di nascita
    - peso
    - sesso
    - altezza
    - allergie
    - intolleranza
    - gusti
    - lavoro/attività fisica
3. app registra l’utente nel repo
4. repo create nuovo profilo utente
5. repo chiede al reasoner delle diete compatibili col profilo utente
6. repo chiede al reasoner il BMI dell’utente
7. app suggerisce un dieta al utente, lo dice se è sovrappeso o no, ecc., “se segui questa dieta, entro un anno sarai ... ", ecc.

3.6 Utente gestisce suoi dati

descrizione

L'utente può aggiornare tutti i dati che ha inserite alle registrazione (vedi il caso precedente). In più, può consultare lo storico dei pasti mangiati, la fedeltà al dieta, ecc.

Descriviamo qua un caso d'uso specifico - si assume che i diversi casi non pongano problemi che sono significativamente diversi

attori:

1. utente
2. app
3. service

scenario

4. Utente apre la app
5. L'utente viene presentato con la *home screen* dell'app.
6. L'utente sceglie "profilo" e poi “peso”
7. Viene presentato con un elenco di pesate storiche, e la possibilità di aggiungerne uno
8. L'utente aggiunge un peso. La data e ora della pesata sono preimpostati per quello di oggi, ma l'utente può cambiarli se vuole
9. L'utente clicca su "salva i dati"
10. L'app manda i dati al orchestrator, che li salva

3.7 Inserimento dati da parte del ristorante

descrizione
Il responsabile di un ristorante registra i piatti del menu nel sistema madiman. Il sistema fornisce il ristorante con dei QR-code per ogni piatto, da stampare sul menu.

attori:
1. ristorante
2. app
3. service
4. nlu

scenario
1. Il ristorante inserisce attraverso la app (forse usando una tastiera) una ricetta nuova in linguaggio naturale
2. Il service chiede al modulo NLU di analizzare la ricetta
3. NLU restituisce la ricetta normalizzata (vedi definizione)
4. L’app chiede conferma al ristorante della ricetta normalizzata
5. Una volta confermato, il sistema restituisce al ristorante un QR-code che identifica il piatto
6. Il ristorante associerà il QR-code al piatto nel menu

3.8 Gestione dati da amministratore

descrizione
L'amministratore è responsabile per le gestione di tutti i dati usati nel sistema del Diet Manager. Gestisce i dati delle ricette, degli alimenti, ma anche degli utente.
Anche se il sistema di gestione è importante per il funzionamento dell'applicazione, è altrettanto vero che l'implementazione di un tale sistema è relativamente semplice. Per cui qua ci limitiamo ad un caso specifico: un amministratore fa il *reset* della password di un utente. Altri casi in cui l'amministratore vuole cambiare, aggiungere o rimuovere dei dati sono analoghi.

attori
- amministratore
- gestionale
- repository

scenario

1. Amministratore apre il suo browser e visita l'URL (per esempio reposity.madiman.org/admin)
2. Il sito chiede login e password, che l'amministratore fornisce

3. Dopo il login, l'amministratore viene presentato con un elenco di "collezioni" da modificare. Può scegliere tra "ricette", "alimenti", "utenti", ecc.
4. L'amministratore clicca sul link "modifica utenti"
5. L'amministratore viene presentato con un elenco di tutti gli utenti, con la possibilità di cercarne dentro
6. L'amministratore cerca per nome, digita "Rossi"
7. L'amministratore viene presentato con 4 risultati trovati, e scegli "Mario Rossi"
8. Appare una scheda con tutti di dati di Mario Rossi
9. L'amministratore clicca sul link "reset password".
10. Il sistema rimuove la password dell'utente dal sistema. Per ragioni di sicurezza, l'amministratore non può settare i password degli utenti - lo devono fare loro stessi, cosi saranno li unici a conoscerla. Il sistema manda, in automatico, un email all'utente con un link dove può reimpostare la password.

3.9 Dietista monitora dieta ad un utente

descrizione
Ogni settimana (o almeno regolarmente) un dietista autonomamente analizza lo storico di un utente e li da dei consigli.

scenario
questo caso d'uso non è stato sviluppato.

3.10 Utente chiede consiglio al dietista

descrizione
L'utente chiede un parere sul andamento di sua dieta al suo dietista

attori

- utente
- dietista
- app
- orchestrator

scenario

1. Utente, dopo aver visualizzato i propri dati, chiede un consulto attraverso l'app. Opzionalmente include una domanda scritta
2. Il service fa sapere ad al dietista dell'utente che l'utente ha bisogno di aiuto
3. Il dietista accede all'app e cerca i dati dell'utente
4. Il dietista risponde sull'app (in lingua naturale), eventualmente cambia la dieta
5. La prossima volta che l'utente aprirà sistema, vedrà la risposta del dietista

4 Repository

In questa sezione studiamo un sistema di repository per le informazioni riguardanti (a) l'utente (b) la sua dieta (c) la ricetta e definiamo dei requisiti di massima

Il repository è responsabile per salvare e rendere a disposizione tutti i dati del sistema. In questa sezione descriviamo alcuni sistemi di *storage*, per poi sceglierne uno sulla base dei requisiti di massima. Dopo descriviamo il *modello dati* - specifichiamo in dettaglio quali sono i dati da salvare nel sistema.

4.1 Sistemi di Storage: stato dell'arte

Il mercato offre decine di sistemi di DBMS ("Database Management System"). Elenchiamo alcuni criteri per scegliere tra le tante alternative.

Una base di dati relazionale o una base di dati noSQL?
In una base di dati relazionale. i dati sono strutturati in tabelle con relazioni tra di loro. Sviluppato negli anni '70, il modello relazionale è quello "classico" per le basi di dati. Tipicamente, dati vengono estratti dal sistema usando il linguaggio SQL ("structured query language"), un linguaggio molto flessibile che permette di "incrociare" le tabelle e creare dei query *ad hoc.* In più, i DB relazionali offrono spesso tanti metodi per assicurarsi del *data integrity* - per assicurare che i dati rimangono consistenti fra di loro.

L'approccio noSQL è relativamente nuova. È un termine che denota una famiglia di approcci diversi al problema di salvare e ricuperare i dati. Tipicamente, un DB noSQL è centrato su "documenti", "key-value store" o "oggetti", un concetto molto più libero e flessibile di quello della "tabella" dei database relazionali. Un grande vantaggio dei database noSQL è che è relativamente facile di *distribuire* i dati - si possono dividere i dati e i calcoli su diverse macchine per ragioni di *availability* a *scalabilità.*

Scalabilità
La scalabilità di un DBMS si riferisce alla possibilità di crescita delle quantità dati e il numero di accessi ai dati senza perdere troppo in termini di velocità. Un DBMS è facilmente scalabile se la base di dati rimane veloce anche se ci sono molti dati, o tanti utenti che devono accedere a questi dati. Una proprietà che incide molto sulla scalabilità è il fatto se è facile distribuire i dati e/o i calcoli su delle macchine diverse.

Availability
L'availablity (cioè la disponibilità) di un DBMS si riferisce al tempo che il DB è accessibile, ed è normalmente espresso in percentuali - una disponibilità del 99% vuol dire che il DB, in media, non è accessibile per 15 minuti al giorno. L'availablity dipende dalla stabilità del sistema -- che in generale, per i DBMS maturi che consideriamo qua, è ottima, ma anche dal fatto se il DB debba essere riavviato per aggiornamenti. La possibilità di distribuire un DBMS su diverse macchine normalmente garantisce una buona *availability:* in caso di instabilità o sovraccarico, i problemi sono in generale limitati ad una singolo macchina, e aggiornamenti che necessitano un riavvio del sistema possono essere fatti separatamente su ogni macchina senza *down time* visibile all'utente.

Security
La *security* (la sicurezza) si riferisce alla integrazione nel DBMS di meccanismi di autenticazione dei utenti, la possibilità di assegnare diritti di visualizzare o cambiare i dati a degli utenti, e il livello di *fine-grainedness* che ha il sistema di permessi.

Distribuzione
Abbiamo già accennato qua sopra il fatto che alcuni DBMS offrano la possibilità di distribuire il DB su diverse macchine. Questa proprietà e essenziale quando la quantità dei dati o accessi diventa così grande che una singola macchina no può gestire tutte le richieste in tempi accettabili. La possibilità di distribuire e duplicare dati su diverse macchine ha anche

Open Source

Tanti DBMS sono rilasciati con delle licenze *Open Source.* Questo vuol dire che il codice del DB è liberamente disponibile e modificabile - normalmente, l'uso dei sistemi open source è gratis. Alcuni DBMS sono proprietari, il che vuol dire che sono a pagamento, e che normalmente il codice non è disponibile per ispezione.

Funzionalità specifica
Alcuni DBMS hanno sopporto nativo per certi tipi di dati o certi tipi di interrogazione dei dati. Per esempio, per i dati geolocalizzati (dati che rappresentano punti o aree su una mappa), alcuni DBMS offrono la possibilità di cercare dati dentro una certa area, o per calcolare la distanza tra due punti.

Nella tabella seguenti analizziamo alcuni sistemi DBMS secondo queste caratteristiche.

	relazionale o no?	scalabilità	availability	security	distribuito	open source	funzionalità geospaziale
postgresql	relazionale	-	+	+	+/-	si	si
mysql	relazionale	-	+	+	no	si	+/-
oracle	relazionale	+	++	++	si	no	si
couchdb	documenti	++	++	+/-	si	si	non nativo
mongodb	documenti	+	+	+	si	si	si
cassandra	key-value	++	++	++	si	si	no

I sei sistemi menzionati nella tabella sono tutti sistemi maturi e usati in progetti molto più grandi di quello che prevediamo per il MadiMan.

Detto questo, dovessimo passare alla implementazione del sistema dopo questo studio di fattibilità, sceglieremmo CouchDB come DBMS. CouchDB combina i punti di forza dei sistemi noSQL con una facilità di istallazione che Cassandra, per esempio, non ha. In più, ha un interfaccia REST nativa, il che vuol dire che l'integrazione con l'orchestrator (che a sua volta implementa un interfaccia REST) è relativamente lineare.

4.2 Modello Dati

In questa sezione descriviamo la struttura dei dati da salvare nel sistema. Ci siamo limitati a descrivere i dati che sono specifiche per MadiMan. Strutture per uso "interno" al sistema - per esempio, dati che hanno a che fare con la sicurezza, come il sistema di permessi - non sono descritti.

4.2.1 Utente

Per gli utenti abbiamo dei dati che vengono usati dal sistema (la password, per esempio), dati anagrafici (nome, etá, ecc.)

Utente	
username	Obbligatorio. Email del utente.
password	Obbligatorio. Una hash del password del

	utente (come è buona pratica, la password stesso non viene salvato per ragioni di sicurezza)
role	Obbligatorio. Il ruolo che l'utente ha nel sistema: può essere un ristoratore, un dietista, utente normale.
creation_date	Obbligatorio. Data di creazione del utente.
name	Obbligatorio. Nome dell'utente.
surname	Obbligatorio. Cognome dell'utente
birth_date	Obbligatorio. La data di nascita del utente. Data normalizzata: YYYY-MM-DD.
sex	Obbligatorio. Il sesso dell'utente. I valori sono della standard ISO 5218: 1 (maschio) 2 (femmina) o 9.
height	L'altezza dell'utente, in metri.
weight	→ storico pesata (4.2.2). Una cronologia di date e peso (in chilogrammi).
work_activity	1 - 5 (quanto è statico il tuo lavoro)
physical_activity	1 -5 (quanta attività fisica fai nel tempo libero)
diet	→ dieta (4.2.3)
meal_history	→ storico pasti (4.2.8). Cosa ha mangiato l'utente, e quando
allergies	→ allergeni 4.2.7
intolerances	→ allergeni 4.2.7
taste	→ alimenti che non piacciono al utente (4.2.5)

4.2.2 Storico Pesata

Lo storico delle pesata registra il peso di un utente ad una certa data

Weight	
user_id	Obbligatorio. ID di un utente.
weight	Obbligatorio. Peso in chili. Reale. Per esempio 76.4

date	Obbligatorio. Timestamp ISO 8601. Data e ora della pesata

4.2.3 Dieta

La dieta è una serie di quantità ideali per ogni nutriente da assumere in un giorno.

La dieta è calcolato per ogni utente, sulla base dei suoi dati, seguendo lo schema elaborato nel file fabbisogni_2013_De_Michieli.xls, che è allegato a questo documento.

Diet	
diet_id	
proteine_min	Intero, in kCal - minimo giornaliero di proteina da assumere
proteine-max	Intero, in kCal - massimo giornaliero
fat_min	Intero, in kCal - minimo giornaliero di lipidi
fat_max	Intero, in kCal - massimo giornaliero di lipidi
carbohydrates_min	Intero, in kCal - minimo giornaliero di carboidrati da assumere
carbohydrates_max	Intero, in kCal - massimo giornierio
proteine_division	Divisione del macro-nutriente sui principali pasti (colazione, pranzo, cena), in percentuali. Per esempio [20,40,40]
fat_division	Divisione del macro-nutriente sui principali pasti
carbohydrates_division	Divisione del macro-nutriente sui principali pasti
alimenti_da evitare	→ alimenti (4.2.5)

VEDI BANCA DATI IEO.xls per una lista più completa di nutrienti da monitorare in una dieta.

4.2.4 Ricetta

Una ricetta contiene titolo, autore, fonte e descrizion, e, per ogni ingrediente, la quantità di quel ingrediente. In più, vengono specificati eventuali "criticità" di preparazione che possono influenzare sulla dieta - per esempio, se la ricetta contiene degli ingredienti fritti, e vengono notati eventuali allergeni presenti nella ricetta.

Recipe	
title	Il titolo della ricetta ("pasta alla carbonara"). Obbligatorio
author	L'autore della ricetta.
source	La fonte della ricetta
description	Descrizione in linguaggio naturale della ricetta.
ingredients	→ alimenti (4.2.5) + quantità (in grammi). Una lista di alimenti e quantità (in grammi) che fanno parte della ricetta.
preparation	modalità di preparazione (4.2.6) per esempio “frittura”, “cottura lunga”. Opzionale. È una lista.
allergens	--> allergeni (4.2.7) Presenza di allergeni. Per esempio "tracce di arachidi".

4.2.5 Alimenti

Gli alimenti sono gli ingredienti di una ricetta, per esempio olio, farina, melanzane. Per ogni alimento abbiamo informazioni sui nutrienti che tipicamente sono contenuti un in alimento. Questa lista è basata sul file BANCA DATI IEO.xls che è allegato. i valori sono in grammi o in milligrammi

Food	
codice	
description	Descrizione corta del alimento.
Categoria merceologica	
Parte edibile	
Acqua	
Proteine totali Proteine animali	
Proteine vegetali	
Lipidi totali	

Lipidi animali	
Lipidi vegetali	
Saturi totali	
Acido oleico	
Monoinsaturi totali	
Acido linoleico	
Altri polinsaturi	
Polinsaturi totali	
Colesterolo	
Gluc.disponibili	
Amido	
Glucidi solubili	
Fibra alimentare	
Alcool	
Energia	in kCal
Ferro	
Calcio	
Sodio	
Potassio	
Fosforo,	
Zinco	
Tiamina	
Riboflavina	
Niacina	
Vitamina C	
Vitamina B6	
Acido folico	
Retinolo eq.	

Retinolo	
β-Carotene	
Vitamina E	
Vitamina D	

4.2.6 Modalità di preparazione

modalità di preparazione	
preparation_id	
name	per esempio “fritto”
description	

4.2.7 Allergeni/Intolleranze

allergens	
allergen_id	
name	
description	

4.2.8 Storico Piatti

Lo storico dei piatti mangiati elenca i pasti consumati da un utente.

Storico Piatti	
date	data di quando è stato mangiato il pasto
time	fascia oraria del pasto. Valori sono: colazione, merenda mattina, pranzo, merenda pomeriggio, cena, fuori pasto
user_id	→ utente (4.2.1)
recipe_id	→ recipe, un link al piatto mangiato (4.2.4)
posto	dove ha mangiato il piatto
quantità	

4.2.9 Ristorante

Ristorante	
restaurant_id	
name	il nome del ristorante
piatti	→ una lista di piatti disponibile in questo ristorante (4.2.4)
street	
postal_code	
city	
country	
telephone	
date_added	

5 Orchestrator

L'orchestrator è responsabile per la comunicazione tra i vari componenti del sistema. Funziona come il fulcro nel eco-sistema di MadiMan. Per facilitare la comunicazione, l'orchestrator rende disponibile un *API (Application Programming Interface*) agli altri moduli, e lo fa tramite un *protocollo di comunicazione*: un insieme di regole formalmente descritte, definite al fine di regolare la comunicazione tra le diverse entità.

Questa sezione include uno studio dello stato dell'arte nei protocolli di rete per comunicazione di web service per dati multimediali strutturati e definizione dei requisiti di massima per i protocolli, ed una descrizione dettagliata dell'API.

5.1 Protocollo REST

Ci sono due cose scelte da fare; (1) decidere sul *protocollo* di comunicazione, cioè il modo in cui i componenti si scambiano informazioni tra di loro, e (2) il *formato dei dati* che vengono scambiati, cioè come vengono rappresentati i dati.

5.1.1 REpresentational State Transfer (REST)

REpresentational State Transfer (REST) è un insieme di linee guida e *best practices* per organizzare servizi web. I sistemi che seguono i principi REST sono spesso definiti "RESTful". I termini "representational state transfer" e

"REST" furono introdotti nel 2000 nella tesi di dottorato di Roy Fielding, uno dei principali autori delle specifiche dell'Hypertext Transfer Protocol (HTTP).

L'idea di base è che si possano usare delle proprietà del protocollo HTTP - cioè il protocollo che usano i browser per navigare su internet - per implementare degli altri tipi di servizi *online*.

Il protocollo HTTP è molto diffuso – è quello usato da tutti i browser per navigare sul web. Il fatto che il protocollo HTTP è così comune rende relativamente facile l'integrazione tra sistemi che devono implementare il protocollo. In più, usando HTTP il servizio può utilizzare tutti I *tool* e gli standard che sono stati sviluppati nel eco-sistema di HTTP; per esempio, si possono usare browser normali per ispezionare I dati, usare servizi di rete per *caching* e *load-balancing*, usare HTTPS per comunicazione encrittato, ecc. ecc..

Principi

REST prevede che la scalabilità del Web e la crescita siano diretti risultati di pochi principi chiave di progettazione:

- Lo stato dell'applicazione e le funzionalità sono divisi in *risorse web .*
- Ogni risorsa è identificato con un URL.
- Tutte le risorse sono condivise come *interfaccia uniforme* per il trasferimento di stato tra client e risorse, questo consiste in:
    - un insieme vincolato di operazioni ben definite
    - un insieme vincolato di contenuti
- un protocollo che è:
    - client-server
    - stateless
    - cachable
    - a livelli

Client–server
L'architture REST si basa su un separazione tra “client” e “server”. Il client manda una richiesta (per informazione, per cambiare dei dati, ecc.); il server risponde (con le informazioni richieste, con l'esito di un'operazione, ecc.)
Questa separazione di ruoli e preoccupazioni significa che, per esempio, il client non si deve preoccupare del salvataggio delle informazioni, che è la responsabilità del server. Il server, invece, non si deve preoccupare dell'interfaccia grafica o dello stato dell'utente. In questo modo i server possono essere più semplici e più facilmente scalabili. Server e client possono essere sostituiti e sviluppati indipendentemente (fintanto che l'interfaccia non viene modificata).

Stateless
La comunicazione client–server è ulteriormente vincolata in modo che ogni richiesta è indipendente da qualsiasi richiesta avvenuta prima: nessun contesto client venga memorizzato sul server tra le richieste. Questo vincolo implica che ogni richiesta da ogni client deve contenere tutte le informazioni necessarie per richiedere il servizio.

Cacheable
Come nel protocollo HTTP, i client possono fare *caching* delle risposte - possono memorizzare certe risposte per motivi di velocità, o perché forse una connessione al server non è disponibile. Il server controlla quale risposte

sono cacheable, e per quanto tempo, in modo da prevenire che i client riusino stati vecchi o dati errati. Una gestione ben fatta della cache può ridurre o parzialmente eliminare le comunicazioni client server, migliorando scalabilità e performance.

Layered system
Un client non può sapere se è connesso direttamente ad un server di livello più basso od intermedio. I server intermedi possono migliorare la scalabilità del sistema con load-balancing o con cache distribuite. Layer intermedi possono offrire inoltre politiche di sicurezza.
Un numero qualsiasi di *connettori* (client, server, cache, tunnel ecc.) può mediare la richiesta, ma, grazie al fatto che il protocollo è *stateless,* ogni connettore interviene senza conoscere la "storia passata" delle altre richieste. Di conseguenza una applicazione può interagire con una risorsa conoscendo due cose: l'identificatore della risorsa e l'azione richiesta - non ha bisogno di sapere se ci sono proxy, gateway, firewalls, tunnel, ecc tra essa e il server su cui è presente l'informazione cercata.

Uniform Interface
Un'interfaccia di comunicazione omogenea tra client e server permette di semplificare e disaccoppiare l'architettura, la quale si può evolvere separatamente.

Risorse
Un concetto importante in REST è l'esistenza di *risorse* (fonti di informazioni), a cui si può accedere tramite un identificatore globale (un URI). Per esempio, una risorsa dove si trova l'informazione di un utente potrebbe avere la forma "http://esempio.com/utenti/mariorossi".

5.1.2 Metodi HTTP

Un concetto importante nel protocollo REST è quello di una *risorsa,* che è identificato con un URI. Per ogni risorsa vengono implementati uno o più azioni. Queste azioni possono essere identificati con uno o più "metodi" o "*verbi*" del protocollo HTTP. In particolare, si distinguono i seguenti verbi.

GET
Il metodo GET tende di recuperare le informazioni localizzate dalla risorsa.
Se la URI della richiesta fa riferimento ad un processo che produce dati, saranno restituiti dalla risposta i dati prodotti da questo processo. Per esempio, una richiesta GET sulla risorsa "/utenti" potrebbe restituire la lista di tutti gli utenti, e un GET sulla risorsa "/utenti/mario_rossi" restituisce tutti i dati dell'utente identificato come mario_rossi.

HEAD
Il metodo HEAD è identico al GET eccetto il fatto che il server non deve ritornare il corpo del messaggio. Questo metodo è usato per ottenere informazioni inerenti all'entità riferita dalla richiesta senza trasferire l'entità stessa. Il metodo può essere usato per testare l'accessibilità di una risorsa, o di vedere se una risorsa è stata modificata.

POST
Il metodo POST è spesso usato per estendere i contenuti del database. Per esempio, una richiesta POST sulla risorsa "/utenti" può essere usato per aggiungere un nuovo utente alla collezione.

PUT
Il metodo PUT aggiornare i dati di una risorsa. Per esempio, un PUT sulla risorsa "/utente/mario_rossi" può essere usato per cambiare, per esempio, l'indirizzo di Mario Rossi.

DELETE
Il metodo DELETE richiede che il server ricevente elimini la risorsa specificata dal URI della richiesta.

5.1.3 Risposte HTTP

Quando il server riceve una richiesta dal client per una risorsa, la risposta consiste di un "body" - l'attuele contenuto della risposta - e un "header", che contiene informazione codificata sullo status delle risposta.

Il Header, per esempio, può dare informazione sulla possibilità di caching, quando è stato modificato la risorsa, di che tipo sono i contenuti, ecc.

Una parte importante del header è lo *status code,* che è un codice a tre cifre che ha la funzione di fornire al client delle informazioni di stato riguardo all'esito della ricezione della richiesta.

La prima cifra dello Status-Code definisce 5 classi di risposta:

-1xx: Informazione - richiesta ricevuta e continuo processo

-2xx: Successo - L'azione è stata ricevuta, capita e accettata

-3xx: Ridirezione - C'è bisogno di altre informazioni per completare la richiesta

-4xx: Client Error - Errori nella richiesta

-5xx: Server Error - Il server fallisce

Elenchiamo i status code più comuni:

CODICE	**FRASE**
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized

402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Time-out
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Large
415	Unsupported Media Type
416	Requested range not satisfiable
417	Expectation Failed
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Time-out
505	HTTP Version not supported

5.2 Formato dati: JSON e XML

Il protocollo HTTP/REST ci fornisce delle linee guide e vincoli per decidere come implementare la comunicazione tra i vari moduli MadiMan. Non specifica come strutturare il contenuto dei messaggi - la scelta di un linguaggio per rappresentare i dati è ancora da fare.

Ci sono diverse alternative per rappresentare dati in un file di testo. Per il nostro servizio, abbiamo scelto di usare JSON, ma riteniamo utile motivare questa scelta facendo un confronto col principale alternativa, quella di XML.

XML (sigla di eXtensible Markup Language) è un linguaggio di markup, ovvero un linguaggio marcatore basato su un meccanismo sintattico che consente di definire e controllare il significato degli elementi contenuti in un documento o in un testo.

In origine, XML è stato pensato per annotare dei testi, per esempio cosi:

```
<frase><persona id="mario_rossi">Mario</persona> abita a Roma</frase>
```

Però, essendo un formato molto flessibile (e anche "estendibile", come dice già il nome), può anche essere usato per rappresentare dei dati molto strutture che non sono necessariamente testi.

```
<?xml version="1.0" encoding="UTF-8"?>
<alimento>
     <codice>22</codice>
     <descrizione>Pane Integrale</descrizione>
     <aqua>36.6</aqua>
     <proteine>7.5</proteine>
     <lipidi>1.3</lipidi>
     <glucidi>53.8</glucidi>
     <energia unit="kcal">243</energia>
</alimento>
```

JSON è un acronimo di *JavaScript Object Notation.* L'esempio di XML qua sopra, in JSON, potrebbe essere rappresentato così:

```
{
     “tipo”: “alimento”,
     "codice": "22".
     "descrizione": “Pane Integrale",
     "aqua": 36.6,
     “proteine”: 7.5,
     “lipidi”: 1.3,
     “glucidi” : 53.8,
      “energia”: 243
}
```

Facciamo un rapido confronto tra i due linguaggi di rappresentazione:

JSON

Pro:

1. JSON ha una sintassi molto semplice, che risulta in una rappresentazione molto più conciso in confronto con XML
2. JSON è facile da usare e integrare, specificamente con Javascript (perché JSON è un sottoinsieme di *JS object literal notation* e usa gli stessi tipi), ma anche con altri linguaggi di programmazione.
3. JSON è integrato in modo molto stretto con sistemi di base di dati noSQL, che usano spesso JSON per la rappresentazione dei dati.

Con:

1. Una sintassi molto semplice vuol anche dire che il linguaggio è meno espressivo - ci sono solo pochi tipi di dati, e poche possibilità di specificare vincoli per garantire la *data integrity.*

XML

Pro:

1. XML ha un markup generalizzato: è possibile creare dei “dialetti” per dei scopi specifici.
2. Esistono tanti tool per la validazione tipi di dati e la struttura del XML, per esempio XML Schema. Rende anche possibile la creazione di nuovi tipi di dati, per esempio SOAP
3. XSLT per trasformare un file XML in un altro formato
4. L'esistenza di XPath/XQuery per estrarre informazione, che rende molto più facile l'estrazione in strutture molto nidificate rispetto a JSON

5. L'esistenza di name spaces, che rende più facile la combinazione di diversi dialetti in un unico documento.

Con:

1. Relativo a JSON, XML è più verboso - per trasmettere la stessa quantità di informazione, usa più caratteri.
2. La complessità e la flessibilità del formato rende più complicata la creazione delle strutture, e l'estrazione dei dati.

In generale, si può dire che XML è probabilmente la scelta più adatta in situazioni in cui la struttura dei dati è molto complessa, o in cui a dati sono testi annotati, o quando si deve lavorare con dati che provengono da fonti diversi che devono essere combinati in un unica struttura.

Per il nostro caso, invece, abbiamo a che fare con dei dati con una struttura relativamente semplice, e riteniamo la scelta per un formato più semplice e “spoglio” il più adatto. Per cui useremo JSON come linguaggio per rappresentare e scambiare i dati.

5.3 IL FRAMEWORK DI SOFTWARE

Deciso il protocollo, la base di dati da usare, e come rappresentare i dati, bisogna ancora decidere come mettere i pezzi insieme. Occorre decidere su un linguaggio di programmazione, e quali framework e librerie usare per implementare il servizio.

Qua, il gamma delle scelte è enorme: ci sono centinaia di framework maturi e buoni in decine di linguaggi di programmazione.

Sarebbe fuori scopo di questo studio di fattibilità di fare un confronto tra tutte le possibilità. Invece, proponiamo qua una possibile configurazione. Ci abbiamo lasciato guidare sopratutto sulla base delle competenze di nostra azienda.

Linguaggio di programmazione: Python

Webserver: Apache via mod_wsgi

Webserver/programming framework: pyramid (http://www.pylonsproject.org/)

Moduli e librerie:

cornice (per implementare un servizio REST): http://cornice.readthedocs.org/en/latest/

couchdb-python (per 'interfaccia con couchdb): https://github.com/djc/couchdb-python

5.4 API REST per il SERVICE

In questa sezione definiamo l'API ("Application Programming Interface") del servizio da sviluppare. Elenchiamo tutti i modi possibili in cui altri moduli del progetto MadiMan possono interagire con l'orchestrator.

Essendo un servizio RESTful, l'API è organizzato attorno un numero di *risorse.*

Ogni risorsa è associato con un URL. Per esempio, se il repository sarà disponibile su repostory.madiman.org, la risorsa “/user” sarà disponibile su http://repository.madiman.org/user.
Per ogni risorsa descritta qua sotto verranno implementati uno o più dei verbi del protocollo HTTP: HEAD, GET, PUT, POST e DELETE.

Il *client* – che potrebbe essere un browser, o l'applicazione del WP1, manda una richiesta secondo il protocollo HTTP al URL della collezione, in cui specifica uno dei verbi, ed eventualmente ulteriori parametri, come specificato nel API. Per esempio, visitare la URL su /user con un browser normale corrisponde con una richiesta GET senza altri parametri.

Il repository risponde con risposta HTTP, in cui sono specificato una riga di stato e il contenuto della risposta. Se la richiesta è andata a buon fine, il server risponse con uno status code di "200 OK" e i dati richiesti nel body.

L'API segue una semplice struttura.

Per ogni tipo di dato – utenti, ricette o alimenti, per esempio – ci sono due tipo di risorsa: una risorsa per gestire la *collezione,* e una per gestire i dati di un singolo elemento delle collezione.

Continuando il nostro esempio, sulla URI "/user" troveremo la collezione, mentre per ogni elemento della collezione (cioè per ogni utente) c'è una risorsa sulla URI "user/12345" dove troviamo i dati del singolo utente. Ogni utente ha un "id" - un numero che identifica inequivocamente l'utente nel repository. I dati del utente con id "12345" si troveranno su /user/12345. In questa risorsa possiamo modificare i dati dell'utente, o rimuoverlo completamente dal db.

Tipicamente, sulla collezione, si possono fare due tipi di richiesta: GET e POST.

Una richiesta GET sulla collezione restituisce un elenco con gli oggetti nella collezione. Questo metodo ha parametri per controllare come sono restituiti i dati - "limit" è un numero che regola il numero di oggeti restituiti, e "start" specifica dove iniziare l'elenco. Tipicamente, saranno implementati un numero di parametri per cercare degli elementi specifici - per esempio, tutti gli utenti che hanno un cognome che inizia con "A", o tutti gli utenti che si sono iscritti nell'utlimo mese. Per esempio, se ci sono 97 utenti, una richiesta GET su /user con limit=10 e start=5 restituisce dati di 10 utenti, iniziando dal quinto trovato. I

Una richiesta POST sulla collezione è usato per aggiungere un nuovo oggetto. Il client deve fornire il repository con tutti i dati obbligatori per l'oggetto. Se i dati sono corretti, il repository risponde con un 200 OK e l'id del nuovo oggetto.

Per la risorsa di un elemento della collezione, l'API tipicamente, tre tipi di richiesta: GET, PUT e DELETE.

Con una richiesta GET si leggono i dati del oggetto; il servizio restituisce un JSON con tutti i dati.

Il verbo PUT è usato per cambiare i dati dell'oggetto. La richieste deve fornire i dati da modificare.

Finalmente, DELETE è utilizzato per rimuovere l'oggetto interamente dalla base di dati.

5.4.1 risorsa /user

GET
Cerca tra gli utenti. Restituisce una lista di user_id che possono essere usati per recuperare i dati dell'utente.

Parametri:

`user_id`: se dato, restituisce l'utente con questo user_id
`name`: cerca utenti con questo nome

`creation_date_from`: Una data in formato ISO. Restituisce utenti creati dopo questa data.
`creation_date_to`
`birth_date_from`
`birth_date_to`
`text`
`order`: ordina i risultati per questo campo.
`limit`: un intero. Specifica quanti risultati restituire. Default è 100.
`start`: un intero.

Risposta

La risposta riassume dati della query, come il numero di risultati trovati, e restituisce una lista di user_id, che possono essere usati per recuperare i dati del utente (con un GET sulla risorsa /`user/{user_id}`)

```
{
  'query_used': {
     'creation_date_from': '2015-01-01',
     'start': 23,
     'limit': 3
  },
  'total_number_of_results': 253,
  'start': 23,
  'number_of_results': 3,
  'results': [
     'userid123',
     'userid23',
     'userid235',
  ],
}
```

POST

Aggiunge un utente nuovo

Parametri
il body delle richiesta contiene una struttura JSON con i dati del utente, che segue la struttura del modello dati 4.2.1

Risposta:
Se la richiesta va a buon fine, restituisce una struttura JSON con tutti i dati del utente che è stato inserito, incluso la user_id creato del sistema.

Esempio:

```
POST /users
{
     "username": "ivan@example.com",
     "password": "are£dfsa",
     "name": "Ivan",
     "surname": "Lunardi",
     "birth_date": "2013-02-13"
}
```

```
200 OK:
{
     "user_id" : "12345"
     "username": "ivan@example.com",
     "name": "Ivan",
     "surname": "Lunardi",
     "birth_date": "2013-02-13"
}
```

```
409 Conflict
{
     "error_code": 123,
     "error_message": "Username already exists"
}
```

```
400 Bad Request
{
     "error_code": 888,
     "error_message": "Invalid date: birth_date"
}
```

```
400 Bad Request
{
  "error_code": 123,
  "error message": "'name' is required"
}
```

5.4.2 risorsa a /user/{user_id}

Risorsa di un utente, identificato dal user_id.

GET

Restituisce i dati dell'utente.

parametri

questo metodo non ha parametri

esempio

```
GET /user/1234

200 OK
{
     "user_id": 1234,
     "username": "ivan@example.com",
     "name": "Ivan",
     "surname": "
     "allergies": [
```

```
        {
        "id": 123,
        "name": "Nichel"
      },

      ….
    ],
    "intolerances": [
      {
        "id": ..
        "name": ...
      },
      ...
    ]
  }
```

PUT

Cambia l'informazione dell'utente.

parametri
Il body delle richiesta contiene una struttura JSON con i dati dell'utente che sono da modificare.
I dati che non vengono specificati rimangono invariati

risposta
Se l'operazione va a buon fine, l'orchestrator risponde con i dati dell'utente, aggiornato come da richiesta

Esempio:

```
PUT /user/12345
{
    "last_name": "Lunardi2",
    "sex": 2,
}

200 OK
{
  "user_id": 1234,
  "username": "ivan@example.com",
  "name": "Ivan",
  "surname": "Lunardi2",
  "sex": 2,
  "allergies": [
          {
            "id": 123,
            "name": "Nichel"
          },

          ….
        ],
  "intolerances": [
          {
            "id": ..
            "name": ...
```

```
        },
        ...
      ]
}
```

DELETE
Rimuove tutti i dati di questo utente dal sistema.

parametri
Questo metodo non ha parametri

risposta
Un messagio che descrive il risultato dell'operazione

esempio

```
DELETE /user/12345

200 OK
     {
      'message': 'User 12345 was succesfully deleted',
     }
```

5.4.3 risorsa a /user/{user_id}/weight

GET
Lo storico del peso dell'utente identificato dal parametro user_id.

parametri:
`date_start`: una data - restituisce solo i pesi avvenuti dope questa data
`date_end`: una data
`limit`: integer

esempio:

```
GET /user/12345/weight

200 OK
[
  {"weight_id": 123567, "weight": 76.3, "date": "2014-06-17T11:36:21Z"},
  {"weight_id": 234347, "weight": 76.1, "date": "2014-06-18T09:56:21Z"}
….
]
```

POST
Aggiunge una pesata allo storico dell'utente identificato dal user_id nel URL.

Parametri:
Il body delle richiesta deve contenere una struttura JSON che specifica:

`weight`: il peso
`date`: la data della pesata. Se la data non è specificato, il valore è la data di oggi.

Risposta:
In caso di successo, la riposta contiene l'id della pesata

Esempio

```
POST /user/12345/weight
{
   'date': '2014-09-09',
   'weight': 65.4
}

200 OK
{
   'weight_id': 98765,
   'message': 'Succesfully added a new weight',
}
```

Aggiungere una pesata per la data di oggi:

```
POST /user/12345/weight
{
     "weight": 78.8
}
```

Aggiungere una pesata per una altra data:

```
POST /user/12345
{
     "weight": 78.8,
     "weight_date": "2014-03-12", \\ date in ISO-8601 format
}
```

5.4.4 risorsa a /user/{user_id}/weight/{weight_id}

La risorsa dove si trova l'informazione della pesata. Una pesata non è modificabile (per cui non è stato previsto una richiesta PUT); si può però rimuovere dallo storico con una chiamata a DELETE

DELETE
Rimuovi la pesata specificata nel URI

Parametri
nessuno

Risposta
un messaggio

Esempio:

```
DELETE /user/12345/weight/9876

200 OK
{
```

```
    'message': 'Successfully deleted weight 9876',
}
```

5.4.5 risorsa a /recipe

La collezione delle ricetta.

GET
Cerca nella ricette.

parametri

`name`: cerca una ricetta per nome
`text`: cerca nei contenuti
`ingredient`: un id di un ingrediente. Restituisce ricette che usano questo ingrediente
`start`: come sopra
`limit`: come sopra

risposta:

informatizione sul numero di risultati e la query usato, e i dati delle ricette trovate

esempio

```
GET /recipe?text=farina&limit=10

200 OK
{
   'number_of_results': 234,
   'query_used': {
        'text': 'farina',
        'limit': 10,
        'start': 1,
      }.
   'results': [
      {
         'recipe_id': 1,
         'name': 'Torta di mele',
      },
      ....
   ]
}
```

POST
Aggiunge una ricetta.

Questa richiesta a due modilità.

1. Si può POSTare una ricetta analizzata, incluso ingredienti, modalità di preparazione, etc. In quel caso la ricette viene semplicemente aggiunta.

2. Se si fornisce solo nome e descrizione, l'orchestrator chiede al modulo NLU di analizzare la ricetta e fornire i dati strutturati.

parametri

Una struttare JSON con i dati della ricetta da aggiungere, secondo il modello dati in sezione 4.2.4.

In modalità 2, solo i parametri `name`, `source` e `description` sono obbligatori.

In modalità 1, anche i parametri `preparation`, `ingredients` e `allergens` devono essere forniti. Questi ultimi parametri prendono come valore una lista di `id`.

risposta

I dati della ricetta aggiunta, incluso il nuovo id creato dal sistema

esempio (ricetta analizzata)

```
POST /recipe
{
      'name': 'Torta di mele',
      'author': 'Sonia',
      'source': 'Cooking for Dummies part IV',
      'description': 'La torta di mele è un dolce classico, ..',
      'preparation': [34, 51],
      'ingredients': [
            {'ingredient_id': 4, 'amount': 700},
            {'ingredient_id': 56, 'amount': 2},
      ],
      'allergens': [6, 8, 45],
}

200 OK
{
      'recipe_id': 56778,
      'name': 'Torta di mele',
      'author': 'Sonia',
      'source': 'Cooking for Dummies part IV',
      'description': 'La torta di mele è un dolce classico, ..',
      'preparation': [34, 51],
      'ingredients': [
            {'ingredient_id': 4, 'amount': 700},
            {'ingredient_id': 56, 'amount': 2},
      ],
      'allergens': [6, 8, 45],
}
```

esempio (ricetta da analizzare)

```
POST /recipe
{
        'name': 'Torta di mele',
        'author': 'Sonia',
        'source': 'Cooking for Dummies part IV',
        'description': 'La torta di mele è un dolce classico, ..',
}

200 OK
{
        'recipe_id': 56778,
```

```
        'name': 'Torta di mele',
        'author': 'Sonia',
        'source': 'Cooking for Dummies part IV',
        'description': 'La torta di mele è un dolce classico, ..',
        'preparation': [34, 51],
        'ingredients': [
             {'ingredient_id': 4, 'amount': 700},
             {'ingredient_id': 56, 'amount': 2},
        ],
        'allergens': [6, 8, 45],
}
```

5.4.6 risorsa a /recipe/{recipe_id}

La risorsa dove si può trovare, modificare o rimuovere i dati di una specifica ricetta.

Per ragioni di spazio, non abbiamo specificato in dettaglio le chiamate, che comunque sono simili a quelli sopra.

GET
Restituisce i dati della ricetta identificata da `recipe_id`

PUT
Cambia i dati delle ricetta identificata da `recipe_id`

DELETE
Rimuove I dati delle ricetta identificata da `recipe_id`

5.4.7 risorsa a /food

La risorsa dove si trova la collezione degli alimenti.

GET
restituisce una lista di alimenti conosciuti dal sistema

POST
aggiunge un alimento

5.4.8 risorsa a /food/{food_id}

GET
Restituisce dati del alimento identificato da `food_id`

PUT
Cambia i dati del alimento identificato da `food_id`

DELETE
Rimuove I dati del alimento identificato da `food_id`

5.4.9 risorsa a /allergen

La collezione di allergeni e intolleranze

GET
Cerca nella collezioni di allergeni

parametri
`name`: cerca allergeno per nome
`allergen_id`
`start`: come sopra
`limit`: come sopra

POST
Aggiunge un allergeno

5.4.10 risorsa a /allergen/{id}

Informazione di un specifico allergeno o intolleranza.

GET

Restituisce i dati del allergeno

PUT

Cambia i dati di un allergeno

DELETE

Rimuove i dati di un allergeno.

5.4.11 risorsa a /history/

Storico dei pasti del utente

GET
restituisce una lista di pasti - quello che sono stati mangiati dall'utente:

parametri:
`user_id`: obbligatorio, l'ID di un utente
`date_start`: restituisce i pasti consumati dopo questa data
`date_end`: restituisce i pasti consumati prima di questa database
`ora`: restituisce i pasti consumati
`start`: come sopra
`limit`: come sopra

risposta:
Una lista di pasti

POST
aggiunge un pasto allo storico di un utente

parametri:
Questo metodo non prende parametri. La richiesta deve contenere i dati del pasto da aggiungere nel body.
risposta:
i dati della ricetta aggiunta
esempio

```
POST
      {
         'user_id': 1234,
         'recipe_id': 543566,
         'date': '2020-08-19',
         'time': 'colazione',
      }

200 OK

      {
         'history_id': 98958923727,
         'user_id': 1234,
         'recipe_id': 543566,
         'date': '2020-08-19',
         'time': 'colazione',

      }
```

5.4.12 risorsa a /history/{history_id}/

GET
Restituisce is dati di un singolo pasto.

PUT
Cambia i dati di un pasto

DELETE
Rimuove i dati di un pasto.

5.4.13 risorsa a /check_recipe

Data una ricetta e un utente, controlla se la ricetta va bene per l'utente.

Questo servizio è solo in lettura, per cui implementa solo il “GET”.

Chiamando questa funziona, l'orchestrator cerca i dati associato con la ricetta specificata, e calcola i contenuti dei macro-nutrienti presenti nella ricetta. Dopo cerca i dati delle dieta dell'utente. Passa questi dati al modulo Reasoner, chiamando la funzione "check_recipe" di paragrafo 5.5.1. La risposta del reasoner viene poi passato

al modulo NLG mediante la funzione decritta in paragrafo 5.5.2, per ottenere un consiglio in linguaggio naturale e un immagine da mostrare all'utente.

GET

parametri:

`user_id`: Obbligatorio, id di un utente
`recipe_id`: Obbligatorio, id di una ricetta

risposta:

giudizio sintetico: un numero da 1 a 5.
giudizio in linguaggio naturale: una stringa
giudizio visuale: un'immagine

esempio

```
GET /check_recipe?user_id=1456&recipe_id=5456

200 OK

{
   "advice": {
      "scale": 3,
      "description": "Don't eat this!!! ",
      "multimedia": {
         "media_type": "image/jpg",
         "uri": "http://some.url/here.jpg"
      }
      },
    "query_used": {
       "recipe_id": 5456,
       "user_id": 145
    }
}
```

5.4.14 risorsa a /diet

Dove si trovano le diete degli utenti

GET

POST

5.4.15 risorsa a /diet/{diet_id}

Dati di una particolare dieta

GET

PUT

DELETE

5.4.16 risorsa a /restaurant

Dati degli ristoranti registrati nel sistema

GET

POST

5.4.17 risorsa a /restaurant/{restaurant_id}

Dati di un ristorante particolare

GET

PUT

DELETE

5.5 API per altri moduli

L'orchestrator comunica con i vari altri moduli via un API loro. Dai casi d'uso viene fuori le necessità di almeno le seguenti funzioni:

5.5.1 REASONER

Il reasoner viene elaborato nel WP5, dove si trova anche una specifica dettagliato dell'algoritmo usato. Il reasoner è responsabile per decidere se una ricetta va bene per l'utente si o no, e dare un giudizio ragionato a riguardo. Espone un API REST che consiste di una singola funzione:

check_recipe
Dato i dati rilevanti di un utente e una ricetta, controlla se la ricetta va bene per l'utente.
Più specificamente, il reasoner restituisce, sulla base di informazione sulla dieta da seguire e l'importo calorico dei macro-nutrienti in una ricetta, per ogni macro-nutriente un giudizio sintetico: un numero da 0 a 5 e una spiegazione.

parametri:
dieta dell'utente:
struttura JSON come da modello dati, consistente di:
- vincoli minimi a massimi per ogni macro-nutriente
- una lista di alimenti da evitare
- per ogni macro-nutriente, delle tolleranze giornaliere, e per ogni pasto

dati del piatto da giudicare:
- per ogni macro-nutriente, il numero di kCal contenuti nel piatto
- una lista di alimenti usati nel piatto

storico pasti dell'utente degli ultimi 7 giorni

risposta
per ogni nutriente:

giudizio sintetico:
un numero da 0 a 5 (0 piatto inconsistente, 5 piatto ideale)
spiegazione:
dizionario, vedi il documento del WP4.

esempio

```
GET check_recipe
{
  'diet': {
     'fat_min': 300,
     'fat_max': 500,
     'fat_division': [20,40,40],
     'fat_tolerance': [15, 20, 20, 20],
     'proteine....': '...',
     ...




}
```

5.5.2 NLG

L'orchestrator passa l'output della precedente funzione "check_recipe" al modulo NLG, che trasforma il giudizio in una risposta per l'utente finale. Il funzionamento del modulo NLG è sviluppato nel WP4. Il modulo NLG, quindi, prende l'output di "check_recipe" per generare una risposta

generate_response
Traduce in linguaggio naturale il giudizio sintetico e la spiegazione

parametri:
l'output del REASONER, cioè per ogni macro-nutriente un giudizio sintetico: un numero da 0 a 5 e una **spiegazione**
la lista di tutti gli alimenti
dati dell'utente (nome, cognome, dati dell'utente, sesso)

risposta
giudizio in linguaggio naturale:
una stringa
giudizio visuale:
un'immagine (un URI)

5.5.3 NLU

Il modulo NLU (Natural Language Understanding) è sviluppato nel WP3. Il modulo implementerà la seguente funzione "analyze_recipe", che data un ricetta in linguaggio naturale, restituisce una struttura JSON che contiene una descrizione formale della ricetta, come descritto nel modello dati.

analyze_recipe
parametri:

`description` obbligatorio. Testo della ricetta
`title`
`author`
`source`

risposta:
una ricetta (vedi modello dati della ricetta per una specifica più precisa)

esempio

```
GET analyze_recipe?description=lunga%20descrizione%20ecc&title=Torta

200 OK
{
   "id_ricetta":"ricetta1",
   "ingredients":{
      "manzo":70,
      "patate_lesse":150
   },
   "modalità":[
      "fritto"
   ],
   "criticità":[
      "fritto"
   ]
}
```

6 Applicazione web per gestione dei dati

I dati salvati nel repository devono sono gestiti da un *amministratore*. Implementiamo una sito web in cui tutti i dati possono essere gestiti, cambiati e aggiornati.
L'applicazione comunica con il repository usando l'API definito qua sopra – e anche la struttura della applicazione segue la struttura del'API.

6.1 Specifiche

Dato la complessità e la grande quantità di dati da gestire, abbiamo scelta di progettare un applicazione per dei browser normali – funzionerà solo bene su dei device con uno schermo abbastanza grande, con un browser con javascript abilitato.

Visto che l'applicazione sarà usato per amministrare i dati, possiamo assumere che l'utente avrà una certa conoscenza dei dati e ha una certa competenza riguardo l'utilizzo di sistemi di content management. Questo ci da la libertà di progettare un interfaccia semplice che segue molto strettamente la struttura dei dati e dell'API.

6.2 Software

La struttura dell'applicazione è una architetture standard *client-server.* Il *server* è semplicemente il repository.

Dati che l'API espone già tutti i metodi per gestire i dati, non c'è bisogno di implementare un ulteriore *software layer* sul lato server: l'applicazione può sviluppato usando solo HTML e JavaScript. In altre parole, il browser comunica direttamente con il repository.

Per la scelta del *software* da utilizzare abbiamo la scelta tra tante librerie JavaScript, e tante *tools* per generare HTML e CSS.

Qua sotto elenchiamo 5 *frame work* Javascript importanti:

- JQuery
- YUI Library
- Dojo Toolkit
- Ext JS
- Angular.js

Ognuno di queste librerie è molto maturo – esiste già da anni ed ha un *user base* grande. Sono diversi tra di loro dal punto di visto di approccio al problema di costruire un applicazione – alcuni di queste librerie, come Jquery, YUI e Dojo, mettono a disposizione tante funzionalità “di base” con cui si possono costruire delle applicazioni. Altre, come Ext JS e Angular.js, sono sistemi con un livello di astrazione più alta: presuppongono delle scelte sul livello dell'architettura che rendono lo sviluppo più facile e rapido, ma al prezzo di meno flessibilità nel risultato finale.

Visto che la presente applicazione è una applicazione CRUD standard, ha senso di scegliere tra gli ultimi framework. Si propone di usare angular.js.

6.3 Interfaccia

L'applicazione per gestire i dati avrà le schermate seguenti:

1. login
2. per ogni collezione, una lista degli elementi della collezione:
    1. Utenti
    2. Ricette
    3. Alimenti
    4. Dieta
    5. Modalità di preparazione
    6. Allergeni/intolleranze
    7. Ristoranti

3. Per ogni elemento di una collezione, una schermata per cambiare i dati di questo elemento

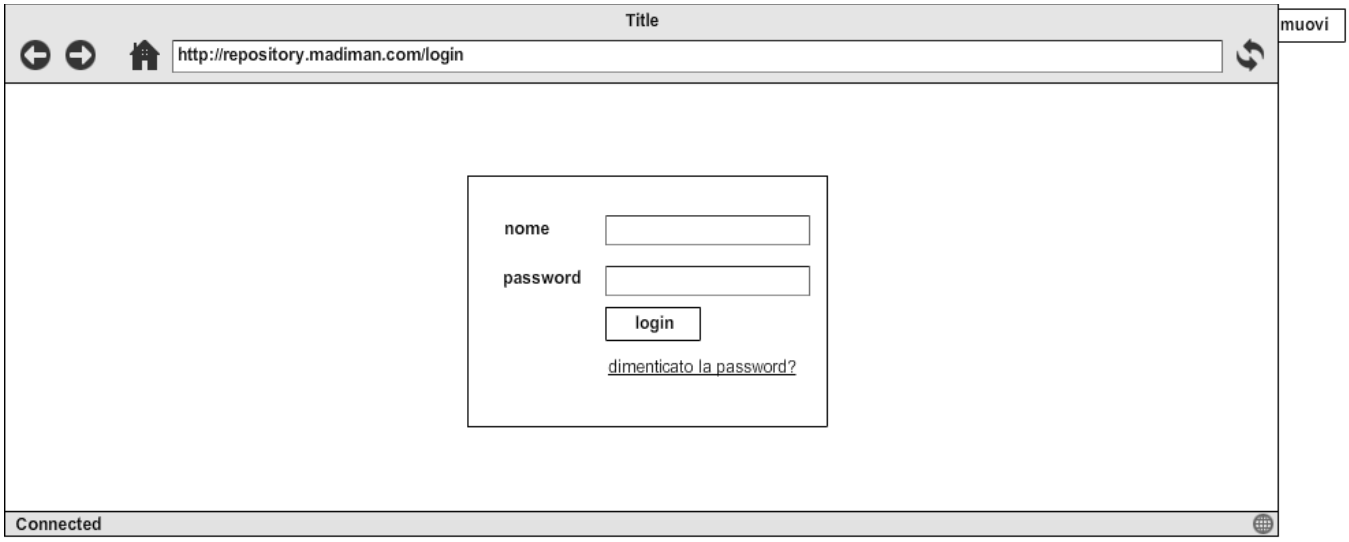
4. Per ogni collezione, una schermate per aggiungere un elemento

5. Una schermata di impostazioni per l'utente, dove, tra l'altro, l'utente può cambiare sua password,

6.4 Wireframe

Riportiamo alcuni "wireframe" per dare un idea come potrebbe essere strutturata l'interfaccia grafica dell'applicazione

La schermata per il login è quello solito:

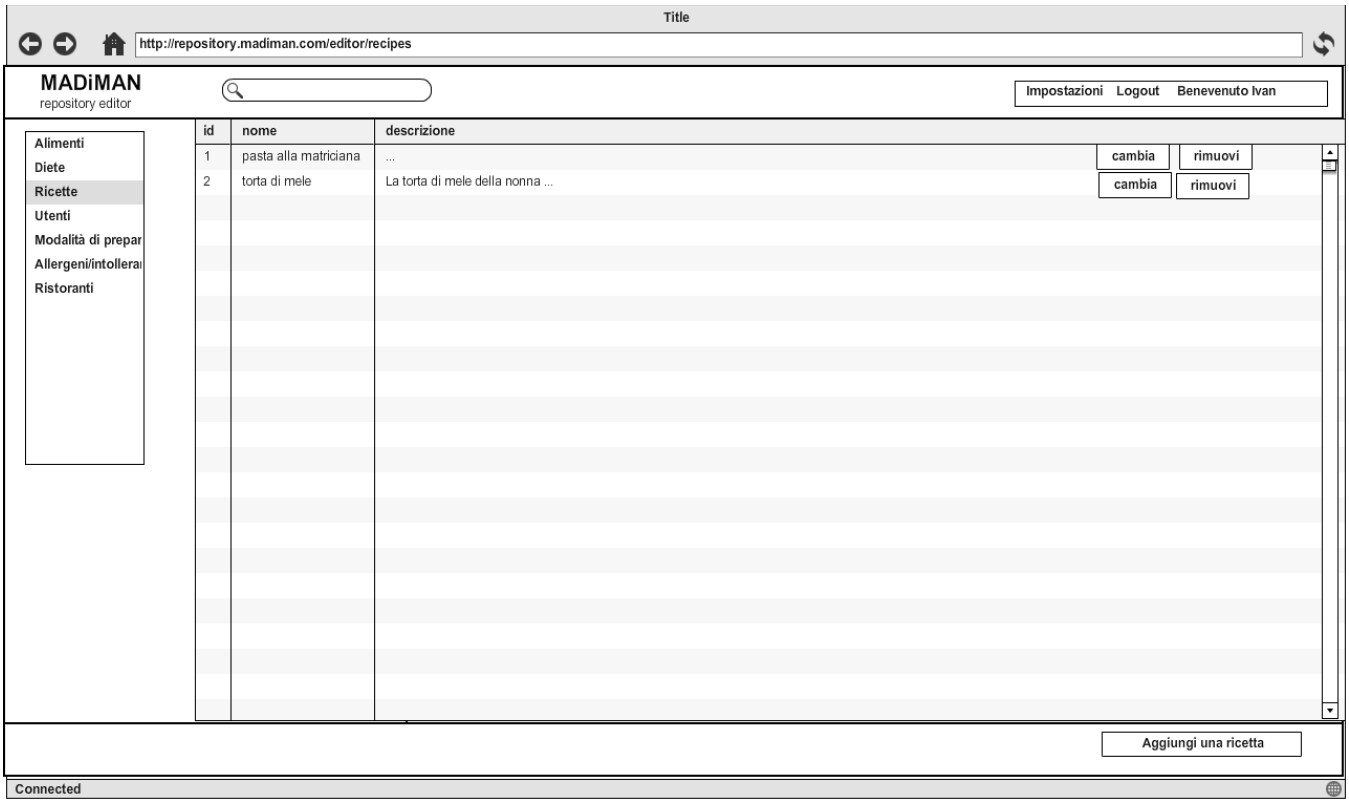


Una volta entra nel sistema, l'utente viene presentato subito con lo scherma principale.

A sinistra si trova la lista di collezioni da modificare. Se l'utente clicca su una collezione (nell'esempio, su "ricetta"), viene presentato con la lista della ricette.

L'utente può ordinare la lista cliccando sulla voce sopra ogni colonna. Può anche cercare dentro la collezione usando la casella di ricerca nella parte superiore della pagina.

Per ogni elemento nella lista si trova un link per cambiare i contenuti, e un link per rimuovere l'elemento dal sistema.



Dopo avver cliccato su "cambia", l'utente viene presentato con una schermata dove può cambiare i dati del elemento - nel nostro esempio i dati di una ricetta per una torta.