AgileClassroom
Una scuola a misura di studente

Antonio Terpin
Christian Bianchini

ITST J.F. Kennedy (Pordenone)

Specializzazione Informatica

'..sot li ciamesutis 'l spirt
di fala uchí la 'Merica..'
(Ornella, 2008)

Anno scolastico: 2016/2017

A tutti coloro che vogliono poter compiere le scelte della loro vita
in base alle loro passioni e ambizioni,
con la consapevolezza che la conoscenza non debba essere un limite,
ma il mezzo.

Ringraziamenti

Questo progetto è stato reso possibile grazie a un percorso lungo tre anni in cui i nostri docenti, alcuni sin dall'inizio e altri solo in quest'ultimo anno, ci hanno permesso di acquisire una mentalità critica e creativa.

In particolare, anche per l'aiuto nella redazione di questo scritto, vogliamo ringraziare le prof.sse Armenio Monica, Fontana Mariachiara, Seno Gloria, Sonzogni Carla e Turchet Cinzia, e i prof. Camilotti Luca, Izzo Leopoldo, Riccio Roberto, Rosace Giulio Umberto, Rossi Giuseppe, Zongaro Gianni e Zuccolo Alessandro.

Inoltre cogliamo l'occasione di augurare il meglio a tutti i compagni di classe che ci hanno offerto numerosi spunti di riflessione e di crescita.

> Ma i veri viaggiatori partono per partire e basta: cuori lievi, simili a palloncini che solo il caso muove eternamente, dicono sempre ≪Andiamo≫, e non sanno perchè. I loro desideri hanno le forme delle nuvole.
>
> *Charles Baudelaire*

Sommario

AgileClassroom è una piattaforma per lo studio che unisce insegnanti e studenti, offrendo una maggiore trasparenza riguardo la situazione della classe, al fine di costrire una scuola a misura di studente. Questo scritto descriverà l'implementazione tecnica e l'idea dietro l'applicazione.

Contenuti

Immagini

Capitolo 1

Introduzione

L'applicazione si basa sull'utilizzo di una *Scrum Board* (Appendice B, Figura B.6): lo studente, per ogni modulo didattico, potrà selezionare le unità di studio dalla sezione *previsti* e posizionarle in quella *da fare*, comunicando l'intenzione di studiare quel particolare concetto nella settimana corrente. Durante lo studio, l'unità sarà posizionata sulla colonna *in studio* e verrà messa in *test* una volta finito. Lì lo studente troverà un numero definito di esercizi, assegnati dall'insegnante, a correzione automatica con un determinato margine di errore, anch'esso a scelta. Superato questo test, segnalerà come *fatta* l'unità, muovendola nell'ultima colonna, mentre in caso di fallimento essa verrà riposizionata nella sezione precedente. In questo modo l'insegnante potrà intervenire aiutando gli studenti con azioni mirate identificando precisamente e tempestivamente le criticità e le lacune di ogni studente, analizzando la *Scrum Board* ed il modo in cui le unità si muovono in essa. Con AgileClassroom chi non è preparato, magari a causa di determinate lacune pregresse, non viene forzato a continuare ad andare avanti senza nemmeno conoscere la destinazione, mentre chi lo è viene stimolato ad approfondire. I docenti, con questo strumento, avranno una visione completa e dettagliata dell'intera classe e potranno gestire le lezioni assegnando del lavoro individuale specifico al fine di ottenere una scuola a misura di studente.

Capitolo 2

Motivazioni e origini

La vita andrebbe vissuta seguendo tutte le proprie passioni e interessi, portandoli avanti al meglio. Vivendo in questo modo, però, si incontrerà un punto critico di oberazione, oltre il quale diventerà complicato gestire tutti gli impegni, ma è proprio una volta superato quel limite che si comprende veramente quanto prezioso sia il tempo (Appendice A). Per questo si è sentita la necessità di ricercare gli sprechi di tempo nell'arco delle ventiquattro ore. Dopo il tentativo fallito con il sonno polifasico, a causa della sua incompatibilità con gli impegni sportivi, e alla riduzione dello spreco di tempo dovuto alle tecnologie, che prolungano la distrazione di una media di 23 minuti e 15 secondi, secondo una ricerca sulle distrazioni digitali (Mark, Gudith, & Klocke, n.d.), il passo successivo è stato quello di analizzare una parte fondamentale della vita di ogni studente: la scuola. Nel frattempo si è avuto modo di entrare in contatto con un insieme di principi e valori nato nell'industria del software ma ora applicato a un largo numero di campi: *Scrum* (Appendice B). Si sono estratti prevalentemente tre concetti dalla lezione delle metodologie agili:

- Gli sprechi sono nocivi e vanno evitati.
- La trasparenza è fondamentale, solo conoscendo i problemi si possono risolvere.
- La felicità è il motore del progresso, oltre che un indicatore predittivo.

Si ha, quindi, provato a comprendere come mai le due fasce laterali, delle tre in cui generalmente viene divisa una classe campione, siano generalmente le più distratte. Prima di tutto, si è notato come i componenti di questi gruppi cambino di materia in materia; la motivazione non risiede pertanto nella personalità dello studente. Abbiamo elaborato una spiegazione che può essenzialmente essere divisa in due parti:

1. Gli studenti che hanno compreso determinati argomenti sono distratti perchè costretti a ripetere concetti ben assimilati.

2. Gli studenti con lacune pregresse, invece, si trovano ad affrontare argomenti di natura più avanzata senza avere le basi per comprenderli.

«Molti scettici potrebbero obiettare: "Sì, grandioso, filosoficamente, questa idea della competenza, l'importanza dell'approccio, gli studenti che studiano con consapevolezza. Ha molto senso, ma non mi sembra molto pratica. Per riuscirci, ogni studente dovrebbe seguire un percorso suo. Dev'essere personalizzata, ci vogliono tutori privati e attività per ogni studente" [..] Ma oggi non è più infattibile.» (Khan, 2015)

Sal Khan, come molti altri, crede in una scuola a misura di studente e porta avanti progetti come la *Khan Academy*. Mentre questo tipo di iniziative si basano esclusivamente sullo studio individuale, l'idea dietro questo lavoro è quella di migliorare l'esperienza scolastica di alunni e insegnanti come gruppo classe, creando uno strumento che possa supportare il docente, fornendogli dei dati e delle informazioni utili per prendere le decisioni più appropriate. Abbiamo deciso di non applicare a pieno *Scrum* alla scuola (come il progetto eduScrum), ma di sfruttare i tre principi prima espressi al fine di creare una classe *agile*.

AgileClassroom mira a semplificare la creazione di percorsi di studio personalizzati per ogni studente, offrendo all'insegnante uno strumento puntuale per il monitoraggio e l'analisi. In questo modo, verrà salvaguardato il tempo di docenti e alunni, evitando di dover seguire corsi di recupero al di fuori dell'orario scolastico. Inoltre, non ci saranno più studenti che si sentono inadeguati a comprendere determinati concetti e che per questo sono portati a prendere decisioni diverse da quelle che vorrebbero.

L'obiettivo di questo progetto è quello di dare la possibilità a tutti gli studenti di realizzare i propri sogni e obiettivi, avendo a disposizione tutte le conoscenze a loro necessarie.

Capitolo 3

Database

3.1 Dati gestiti

Data l'applicazione da implementare (visibile al capitolo 5), si individuano i seguenti elementi:

- **Keychain**: Rappresenta il "portachiavi" dell'utente;
  contiene i dati utili all'autenticazione, ovvero la password (sha256) e il *nonce*, un codice semicasuale di 16 caratteri. L'accesso viene effettuato confrontando *digest* della password, mentre le sessioni sono garantite da un *token* ottenuto tramite la cifrazione con AES in CBC mode dello user_id (l'identificativo dell'utente possessore del portachiavi) e del *nonce*. La scadenza del *token* può quindi essere anticipata cambiando il *nonce*. L'importanza di questo è dovuta non solo a quest'ultima funzione ma anche al fatto che sia esso a determinare la variabilità e l'impredicibilità del *token*, insieme alla chiave usata nella crittazione. Si opta per una cifratura (invertibile), poichè nel *token* in futuro potranno essere inserite informazioni utili, come la posizione da cui è stato effettuato l'ultimo login, per favorire la difesa da furti di sessione. *Keychain* è in corrispondenza biunivoca con *User*.

- **User**: Rappresenta l'utente nel sistema;
  contiene i dati biografici e di contatto dello stesso, come username, email, nome, cognome, data di nascita e percorso sul server dell'immagine profilo. Username e email sono utilizzati per il login e per il recupero della password e sono univoci per ogni utente, ma non sono utilizzati come chiave primaria sia per motivi di comodità e di universalità della query, che per favorire l'anonimato e la privacy (un id trasporta per il malintenzionato che legge molta meno informazione di uno username o un'email), che per mettersi al riparo da eventuali aggiornamenti

e nuove necessità (per esempio sarebbe complicato permettere di cambiare lo username se questo fosse chiave primaria).

- **Subject**: Rappresenta una materia di studio ed è associata ad una propria classe. Ha un programma e può essere completata (anno/periodo di studio concluso) o in corso (attiva). Ha un unico insegnante, il creatore (*User*) della materia.

- **Student**: Associa uno studente ad una materia come studente. L'insieme degli utenti associati in questo modo rappresenta la classe. La coppia *student_id-subject_id* rappresenta una chiave candidata, l'id viene utilizzato per semplicità nelle query e per permettere possibili aggiornamenti. I campi *left* e *join_again* servono a memorizzare il fatto che un utente sia uscito o meno dal gruppo classe e che voglia o meno essere riaggiunto. Questa funzionalità tiene conto della possibilità di utilizzare questa applicazione in contesti diversi da quello dell'istruzione pubblica, come altri corsi di formazione.

- **Module**: Rappresenta il modulo didattico di una particolare materia, ha un nome e una descrizione.

- **Unit**: Rappresenta l'unità didattica di un particolare modulo, ha un nome e una descrizione. Inoltre, ogni unità ha un numero a scelta di esercizi da proporre allo studente che la ponga nella sezione di *test* e un margine di errori da non superare per poter segnalare come compresa l'unità (*numberOfEx*, *errorMargin*).

- **Position**: Definisce per ogni unità (*unit_id*) di una materia la posizione che ha nella *Scrum Board* (*cur_column*) di ciascuno studente (*student_id*). Per quanto la coppia *student_id-unit_id* sia univoca, per comodità ci si appoggia ad un id.

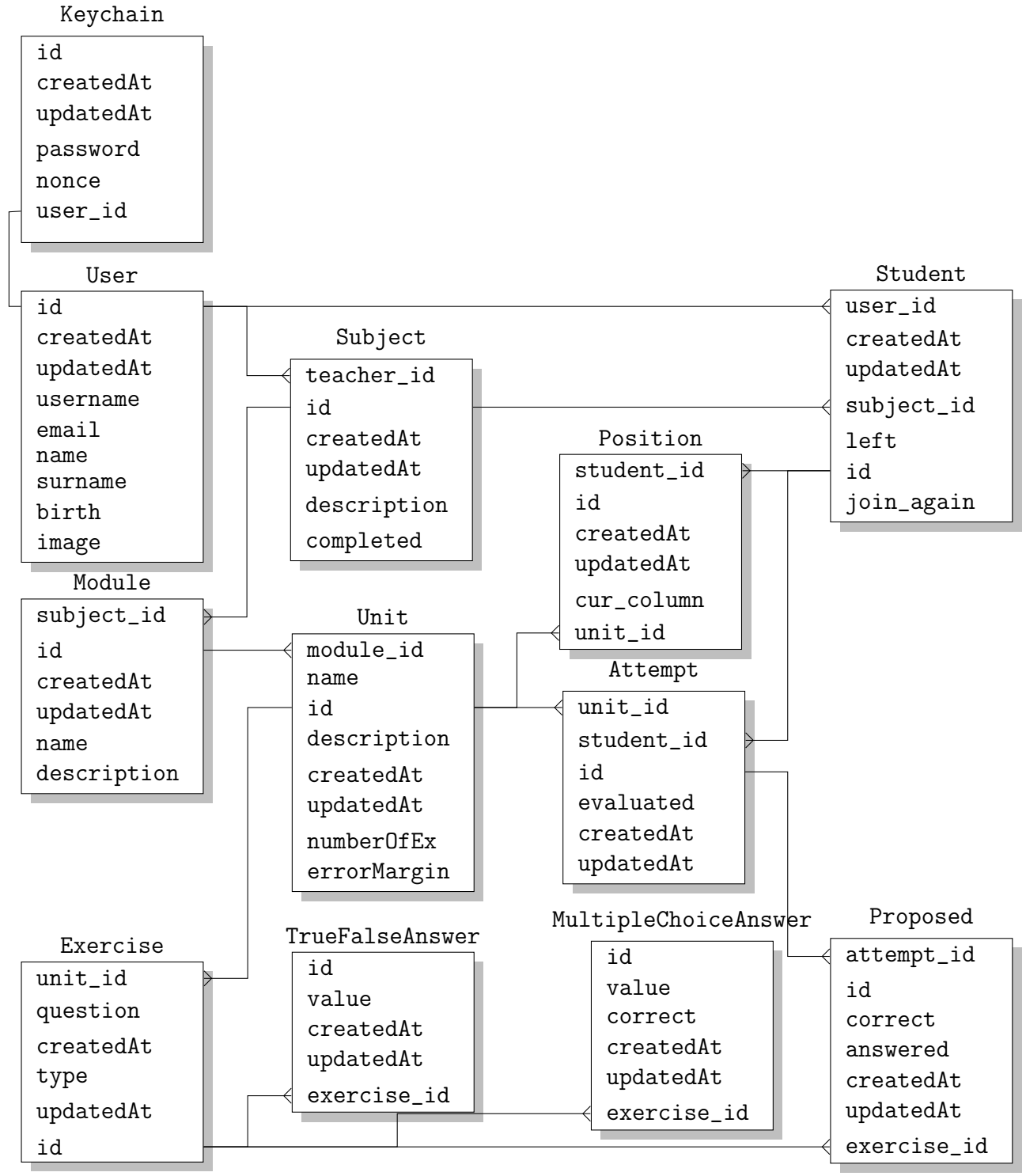
- **Exercise**: Rappresenta un generico esercizio che può essere di tre tipi: a completamento (in questo caso la soluzione è contenuta direttamente nella domanda), a risposta multipla o vero o falso. Ha una domanda ed è associato ad una particolare unità. L'insieme degli esercizi di un'unità rappresenta il *dataset* da cui viene estratto il sottoinsieme di cardinalità scelta da proporre allo studente che ponga l'unità in fase di test.

- **TrueFalseAnswer**: è la risposta (*value*) vera o falsa a un esercizio di tipo vero o falso.

- **MultipleChoiceAnswer**: è una delle opzioni per un'esercizio a scelta multipla. È l'opzione corretta se *correct* vale *true*.
- **Attempt**: viene generato ogni qualvolta si provi a testare un'unità. Se valutato (*evaluated*) può essere superato o fallito.
- **Proposed**: è un esercizio proposto in un *attempt*. Può avere già una risposta (*answered*) e può essere o meno corretto (*correct*).

Ogni entità ha inoltre due attributi (*createdAt*, *updatedAt*), che sono necessari per effettuare correttamente la sincronizzazione del database centrale con quelli locali delle applicazioni consumer (Capitolo 7).

3.2 Schema Database



Per comodità le entità sono nella forma di una tabella comprensiva di tutti i campi della stessa. L'ER è anche indicativo del modello logico.

3.3 Modello fisico

Keychains						
Attributo	Tipo	Dim	Unic	Default	Chiave	Descrizione
id	long		✓	auto-inc	Primaria	Identificativo
createdAt	date			on_insert: now()		Data creazione
updatedAt	date			on_update: now()		Data ultimo aggiornamento
password	char	64				hash (sha256) password utente
nonce	char	16				Per creazione diversi token
user_id	long		✓		Ext: User-id	Utente

Users							
Attributo	Tipo	Dim	Unic	Default	Opz	Chiave	Descrizione
id	long		✓	auto-inc		Primaria	Identificativo
createdAt	date			on_insert: now()			Data creazione
updatedAt	date			on_update: now()			Data ultimo aggiornamento
username	varchar	100	✓				username utente
email	varchar	100	✓				email utente
name	varchar	50			✓		nome utente
surname	varchar	50			✓		cognome utente
birth	date				✓		data di nascita utente
image	varchar	200			✓		percorso immagine utente

Subjects					
Attributo	Tipo	Unic	Default	Chiave	Descrizione
id	long	✓	auto-inc	Primaria	Identificativo
createdAt	date		on_insert: now()		Data creazione
updatedAt	date		on_update: now()		Data ultimo aggiornamento
description	text				Programma della materia
completed	boolean		false		Materia completata
teacher_id	long			Ext: User-id	Insegnante della materia

Students					
Attributo	Tipo	Unic	Default	Chiave	Descrizione
id	long	✓	auto-inc	Primaria	Identificativo
createdAt	date		on_insert: now()		Data creazione
updatedAt	date		on_update: now()		Data ultimo aggiornamento
left	boolean		false		Uscito dalla classe
join_again	boolean		true		Consenso di essere aggiunto nuovamente
subject_id	long			Ext: Subject-id	Materia
user_id	long			Ext: User-id	Studente della materia

Modules						
Attributo	Tipo	Dim	Unic	Default	Chiave	Descrizione
id	long		✓	auto-inc	Primaria	Identificativo
createdAt	date			on_insert: now()		Data creazione
updatedAt	date			on_update: now()		Data ultimo aggiornamento
name	varchar	100				Nome modulo didattico
description	text					Descrizione modulo didattico
subject_id	long				Ext: Subject-id	Materia

Units						
Attributo	Tipo	Dim	Unic	Default	Chiave	Descrizione
id	long		✓	auto-inc	Primaria	Identificativo
createdAt	date			on_insert: now()		Data creazione
updatedAt	date			on_update: now()		Data ultimo aggiornamento
name	varchar	100				Nome unità didattica
description	text					Descrizione unità didattica
module_id	long				Ext: Module-id	Modulo didattico
numberOfEx	int			10		Numero di esercizi da proporre
errorMargin	int			2		Numero di errori massimo

Positions						
Attributo	Tipo	Dim	Unic	Default	Chiave	Descrizione
id	long		✓	auto-inc	Primaria	Identificativo
createdAt	date			on_insert: now()		Data creazione
updatedAt	date			on_update: now()		Data ultimo aggiornamento
cur_column	enum	”Previsti” ”Da fare” ”Studio” ”Test” ”Fatti”		”Previsti”		Colonna della *Scrum Board*
student_id	long				Ext:Student-id	Studente
module_id	long				Ext: Module-id	Modulo didattico

Exercises						
Attributo	Tipo	Dim	Unic	Default	Chiave	Descrizione
id	long		✓	auto-inc	Primaria	Identificativo
createdAt	date			on_insert: now()		Data creazione
updatedAt	date			on_update: now()		Data ultimo aggiornamento
type	enum	”MultipleChoice” ”Completion” ”TrueFalse”		”TrueFalse”		Colonna della *Scrum Board*
question	text					Domanda dell’esercizio
unit_id	long				Ext: Unit-id	Unità didattica

TrueFalseAnswers						
Attributo	Tipo	Dim	Unic	Default	Chiave	Descrizione
id	long		✓	auto-inc	Primaria	Identificativo
createdAt	date			on_insert: now()		Data creazione
updatedAt	date			on_update: now()		Data ultimo aggiornamento
value	boolean			True		Risposta
exercise_id	long	100			Ext: Exercise-id	Esercizio

MultipleChoiceAnswers					
Attributo	Tipo	Unic	Default	Chiave	Descrizione
id	long	✓	auto-inc	Primaria	Identificativo
createdAt	date		on_insert: now()		Data creazione
updatedAt	date		on_update: now()		Data ultimo aggiornamento
value	text				Opzione
correct	boolean		True		Opzione corretta
exercise_id	long			Ext: Exercise-id	Esercizio

Attempts					
Attributo	Tipo	Unic	Default	Chiave	Descrizione
id	long	✓	auto-inc	Primaria	Identificativo
createdAt	date		on_insert: now()		Data creazione
updatedAt	date		on_update: now()		Data ultimo aggiornamento
evaluated	boolean		False		Test finito
student_id	long			Ext: Student-id	Studente
unit_id	long			Ext: Unit-id	Unità didattica

Proposeds						
Attributo	Tipo	Unic	Default	Opz	Chiave	Descrizione
id	long	✓	auto-inc		Primaria	Identificativo
createdAt	date		on_insert: now()			Data creazione
updatedAt	date		on_update: now()			Data ultimo aggiornamento
correct	boolean			✓		Esercizio svolto correttamente
answered	boolean		False			Risposta data
exercise_id	long				Ext: Exercise-id	Esercizio
attempt_id	long				Ext: Attempt-id	Test (tentativo)

3.4 Normalizzazione

1. Il database è in PRIMA FORMA NORMALE, infatti sono rispettati i 5 principi della Relazione teorizzati da Codd:

   - Tutte le righe di ogni tabella hanno lo stesso numero di colonne.
   - Gli attributi rappresentano informazioni elementari secondo il contesto (quindi per tutte e due le accezioni individuate dal matematico per il termine *atomico*).
   - Tutti gli elementi che compaiono in una colonna appartengono allo stesso dominio (omogeneità).
   - Ogni riga è diversa da tutte le altre (esiste almeno una chiave candidata).
   - L'ordine delle righe non è rilevante.

2. Essendo in PRIMA FORMA NORMALE e non avendo in nessuna tabella chiavi composte, il database è anche in SECONDA FORMA NORMALE.

3. Date le dipendenze funzionali:

   - **Keychain**
     $id \rightarrow (createdAt, updatedAt, password, nonce, user\_id)$
     $user\_id \rightarrow (createdAt, updatedAt, password, nonce, id)$
   - **User**
     $id \rightarrow (createdAt, updatedAt, username, email, name, surname, birth, image)$
     $username \rightarrow (createdAt, updatedAt, email, name, surname, birth, image, id)$
     $email \rightarrow (createdAt, updatedAt, username, name, surname, birth, image, id)$
   - **Subject**
     $id \rightarrow (createdAt, updatedAt, teacher\_id, description, completed)$
   - **Student**
     $id \rightarrow (createdAt, updatedAt, user\_id, subject\_id, left, join\_again)$
     $user\_id, subject\_id \rightarrow (createdAt, updatedAt, left, join\_again, id)$
   - **Module**
     $id \rightarrow (createdAt, updatedAt, subject\_id, name, description)$
   - **Unit**
     $id \rightarrow (createdAt, updatedAt, module\_id, name, description, numberOfEx, errorMargin)$

- **Position**
  $id \rightarrow (createdAt, updatedAt, student\_id, unit\_id, cur_column)$
  $student\_id, unit\_id \rightarrow (createdAt, updatedAt, id)$
- **Exercise**
  $id \rightarrow (createdAt, updatedAt, unit\_id, type, question)$
- **TrueFalseAnswer**
  $id \rightarrow (createdAt, updatedAt, exercise\_id, value)$
- **MultipleChoiceAnswer**
  $id \rightarrow (createdAt, updatedAt, exercise\_id, value, correct)$
- **Attempt**
  $id \rightarrow (createdAt, updatedAt, unit\_id, student\_id, evaluated)$
- **Proposed**
  $id \rightarrow (createdAt, updatedAt, attempt\_id, exercise\_id, answered, correct)$
  $attempt\_id, exercise\_id \rightarrow (createdAt, updatedAt, answered, correct, id)$

Si nota che nelle tabelle *Users*, *Keychains*, *Student*, *Proposed* e *Position* si hanno più dipendenze funzionali ma in tutte il determinante rappresenta una chiave (candidata) per la rispettiva relazione. Inoltre il database è in SECONDA FORMA NORMALE, pertanto è anche in TERZA FORMA NORMALE.

Capitolo 4

Web Service

4.1 Descrizione del servizio

Il *backend* dell'applicazione è scritto interamente in PYTHON e si basa sul micro-framework per lo sviluppo web *Flask*(*Flask*, n.d.). È un *web service* di tipo REST che viene gestito dal web server *Apache*. Per tutte le API, fatta eccezione per quelle di login, registrazione e quella di descrizione generale del servizio, è necessario essere autenticati al fine di garantire la *privacy* dei dati. Per facilitare il *deployment* del servizio sono stati scritti due script, che vanno eseguiti con i privilegi di *root*:

- **setup.sh**: esegue ordinatamente le seguenti operazioni:
  1. Installa il DBMS POSTGRESQL.
  2. Installa PIP (*Python Package Index*), ovvero il gestore dei pacchetti di Python.
  3. Crea un utente sulla macchina a cui verrà associato un account *root* su POSTGRES.
  4. Crea tre , per lo sviluppo, per il test e per la produzione e associa all'utente prima creato tutti i privilegi di accesso al *database*.
  5. Permette l'accesso a qualunque di questi *database* con qualunque utente da qualunque ip con protocollo di trasporto TCP.
  6. Riavvia il DBMS per rendere effettive le modifiche.
- **requirements.pip**: permette di installare i package aggiuntivi di PYTHON richiesti dal *web service* per funzionare, tra cui:
  - *flask*, il micro-framework web utilizzato.

- psycopg2, che offre le API di accesso a *PostreSQL*.
- flask_sqlalchemy, l'ORM (*Object Relational Mapping*) di *flask*. Un ORM fornisce un'interfaccia orientata agli oggetti per l'accesso ad un RDBMS, astraendo l'accesso effettivo allo stesso.
- itsdangerous, passlib per *utility* di sicurezza, come il calcolo di *digest* o la cifratura dei dati.

4.2 Ambiente di sviluppo

Per lo sviluppo ci si è fortemente appoggiati alla virtualizzazione ed in particolare si è usato l'ambiente di sviluppo *Vagrant*. Questo consiste in una *utility* a riga di comando che permette di gestire il ciclo di vita delle macchine virtuali. Operando in questo senso si ha costituito un ambiente isolato per lo sviluppo, facilmente replicabile ma indipendente dal sistema operativo: la peculiarità di *Vagrant* è data dal fatto che venendo eseguito a linea di comando una volta posizionati su un determinato ramo del *file system*, si può operare su quegli stessi documenti lavorando sulla macchina reale ma testare quanto fatto come se si operasse su una macchina remota. Per maggiori informazioni riguardo l'utilità ed il funzionamento di questo strumento si rimanda al sito ufficiale. Le specifiche di *Vagrant* sono definite nel file *Vagrantfile*, dove è indicata la *box* utilizzata (*ubuntu/trusty32*) e le informazioni di networking (*public_network*, ip: 192.168.10.160). Si è inoltre usufruito del sistema di *version control* remoto GITlab.

4.3 Flask

Come introdotto sopra, il *web framework* utilizzato è *Flask*. L'inizializzazione e l'avvio dell'applicazione sono strutturati attraverso tre file:

- **run.py**: permette di avviare il servizio, è molto semplice e si limita a richiamare l'inizializzazione dell'applicazione e avviarla (Figura 4.1).

```
# init application
import ac
ac.create_app("development")

# starting application
from ac import app
app.run(host = app.config["HOST"], port = app.config["PORT"],\
        debug = app.config["DEBUG"])

```

Fig. 4.1: Avvio dell'applicazione (*run.py*)

- **ac/__init__.py**: Importando il modulo *ac* si include quanto contenuto in questo file, pertanto la funzione `create_app` e le due variabili `app` e `db`. `create_app` fa riferimento al file *ac/config.py* per ricavare le informazioni di configurazione per istanziare l'oggetto di tipo *Flask* e assegnarlo alla variabile globale `app` (Figura 4.2).

```
def create_app(config_name)
    from flask import Flask
    from config import config

    global app

    app = Flask(__name__)
    app.config.from_object(config[config_name])

```

Fig. 4.2: Istanza di *app*

La funzione del parametro `config_name` è quella di definire il tipo di applicazione desiderata (*development*, *testing*, *production*). Successivamente viene istanziato l'ORM dell'applicazione e create le tabelle (se non presenti) del *database*. Vengono inoltre attivate le *blueprint* (*controllers* o API del *backend*) (Figura 4.3).

```
        global db
        db = SQLAlchemy(app)

        # importing models
        from auth.models import User, Keychain
        from subject.models import Subject
        from student.models import Student
        from module.models import Module
        from unit.models import Unit, Position
        from test.models import Exercise, TrueFalseAnswer,\
                MultipleChoiceAnswer, Attempt, Proposed

        # creating db
        db.create_all()

        # after creating the db connect blueprints
        config[config_name].init_app(app)

```

Fig. 4.3: Istanza di *db* e connessione *blueprints*

Infine vengono definiti i gestori delle eccezioni (*exceptions handlers*). In particolare, in figura 4.4 sono rappresentati i due gestori delle eccezioni del servizio; questi due intervengono quando ne vengono generate di tipo `ACException` e `SQLAlchemyException` e costruiscono una risposta per il *client*.

```
        # ac exception handler
        from ac.exceptions import ACException
        @app.errorhandler(ACException)
        def handleACException(error):
            response = jsonify(error.to_dict())
            response.status_code = error.status_code
            return response

        # SQLAlchemy exceptions handler
        from sqlalchemy.exc import SQLAlchemyError
        from ac.exceptions import throw_exception
        @app.errorhandler(SQLAlchemyError)
        def handleACDBException(error):
            return jsonify(message = 'Errore generico',\
                status_code = 500)

```

Fig. 4.4: Gestori delle eccezioni

Mentre le seconde si riferiscono a errori durante operazioni riguardanti il *database*, per esempio quando vengono violati dei vincoli, e vengono mascherate da *Internal Server Error* generico per evitare *blind attacks*, le prime vengono generate volontariamente dal servizio. Infatti possono essere dovute a un accesso non autorizzato, alla mancanza di permessi o di parametri nella richiesta o a un tentativo di *exploit* (in questo caso l'errore viene mascherato da errore

generico). Questo tipo di eccezioni è descritto dal file *ac/exception.py* e deriva dalla classe `ACException` (Figura 4.5).

```
class ACException(Exception):
    def __init__(self, status_code = 400, message = None, **parameters):
        self.status_code = status_code
        self.message = message
        self.parameters = parameters
        Exception.__init__(self, self.to_dict())

    def to_dict(self):
        rv = {k: v for k, v in self.parameters.items()}
        rv['message'] = self.message or 'Bad request'
        rv['success'] = False
        rv['status_code'] = self.status_code
        # error eventually replaces message
        if 'error' in rv.keys():
            rv['message'] = rv['error']
            del rv['error']
        return rv

```

Fig. 4.5: Eccezione generica: ACException

La classe `ACException` eredita da `Exception`, e richiama il costruttore della classe padre passandogli un array associativo con le chiavi `message` (messaggio da comunicare), `success` (riuscita o meno della richiesta) e `status_code` (codice dell'eccezione) ricevute dalle classi figlie o dal costruttore. In questo modo il gestore dell'eccezione potrà tornare al client in formato JSON l'errore emerso.

- **ac/config.py**: contiene le informazioni di configurazione dell'applicazione, in particolare:
  - La *root directory* dell'applicazione.
  - Le classi di configurazione, ovvero:
    - **Config** (Figura 4.6): La classe di configurazione di base, contenente le chiavi di sessione e di prevenzione da CSRF (*Cross Site Request Forgery*), omesse dal listato, la porta, l'host e le informazioni pubbliche dell'applicazione.

```
class Config:
        PORT = 12345
        HOST = '0.0.0.0'
        # PUBLIC INFOS
        NAME = "AgileClassroom Webservice"
        VERSION = 1.0,
        TYPE = "REST"
        LAST_RUN = datetime.now(pytz.utc)
        # init method
    @staticmethod
    def init_app(app):
        #blueprints
        from base.controllers import base
        from auth.controllers import auth
        from subject.controllers import subject
        from student.controllers import student
        from module.controllers import module
        from unit.controllers import unit, position
        from test.controllers import exercise, attempt

        # register blueprints
        app.register_blueprint(base)
        app.register_blueprint(auth)
        app.register_blueprint(subject)
        app.register_blueprint(student)
        app.register_blueprint(module)
        app.register_blueprint(unit)
        app.register_blueprint(position)
        app.register_blueprint(exercise)
        app.register_blueprint(attempt)

```

Fig. 4.6: Classe generica di configurazione

La classe `Config` contiene inoltre un metodo statico (che in PYTHON, a differenza di altri linguaggi, ha due accezioni; un metodo può essere definito come *classmethod* e ricevere come primo parametro implicito la classe in cui è definito, o come *staticmethod* e non riceverlo ma essere utilizzabile comunque sia tramite un'istanza della classe che tramite la classe stessa) `init_app`, che importa e registra le *blueprint*, rendendole note a *Flask*. Una *blueprint* in PYTHON consiste in un *template* per un modulo di un'applicazione web. Nella sezione 4.4 sono descritti in dettaglio i moduli di AgileClassroom.

* `DevelopmentConfig`, `TestingConfig`, `ProductionConfig`, ovvero classi specializzate di `Config` che definiscono il *database* al quale connettersi (sviluppo, test o produzione) e la modalità di utilizzo dell'applicazione (*debug*, *testing*) utili a Flask per operare in modo specializzato. Per esempio in fase di *debug* è attivo l'*auto-reload* in caso di modifiche al *backend*.

– Un array associativo che permette di accedere alle diverse classi specializzate (Figura 4.7)

```
config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,

    # default configuration
    'default': DevelopmentConfig
}

```

Fig. 4.7: Array associativo per la scelta della classe di configurazione appropriata

4.4 Moduli

In questa sezione si tratta la questione modularizzazione del *web service*. Ogni modulo conterrà un file *controllers.py* in cui sono presenti le *blueprint* per l'accesso a quella sezione. Una *blueprint* è il *template* di una sezione di una applicazione web (Figura 4.8), è caratterizzata da un nome e da un prefisso e permette di appendervi i diversi *endpoint*.

```
1  student = Blueprint("student", __name__, url_prefix = "/student")
2
3  @student.route("/add", methods=["POST"])
4  @needs_parameters("POST", "subject_id", "user_id")
5  @needs_authentication
6  @teacher_zone("POST")
7  def addStudentToClass():
8      student = addStudentToClassAPI(g.post.get("subject_id"), g.post.get("user_id"))
9      return jsonify(student = student.json)
10
```

Fig. 4.8: Definizione della *blueprint* studente e dell'*endpoint ../add [POST]*

Ogni *endpoint* può essere caratterizzato da uno o più *decorators*. Un *decorator* è essenzialmente una funzione che ne modifica un'altra. Ne definisce quindi una nuova al suo interno, che sarà anche il suo risultato. Questa funzione interna performa alcune operazioni influenzando quindi i risultati della funzione "decorata". I *decorators* utilizzati in AgileClassroom sono quelli di autorizzazione (Capitolo 8), quello per estrarre i parametri necessari dalla richiesta dell'utente (Figura 4.9) e quello che obbliga il *consumer* a essere autenticato (Figura 4.10).

```python
def needs_parameters(method, *parameters_keys):
    def decorator(function):
        @wraps(function)
        def decorated_function(*args, **kwargs):
            missing = []
            # where are parameters stored?
            store = storeForMethod(method)
            # key where the found parameters will be put
            outputKey = outputKeyForMethod(method)
            output = {}
            # for each parameter needed
            for key in parameters_keys:
                # is store an array? look for parameter there
                missing_on_array = isinstance(store, list)\
                                   and key not in store
                # is store a dictionary? look for parameter there
                missing_on_dict = isinstance(store, dict)\
                                  and store.get(key) == None
                # is the parameter an empty string?
                empty_string = isinstance(store, dict)\
                               and isinstance(store.get(key), basestring)\
                               and not store.get(key)
                if missing_on_dict or missing_on_array or empty_string:
                    missing.append(key)
                else:
                    output[key] = store[key]
            # setting found values in g at the key output key
            setattr(g, outputKey, output)
            if missing:
                # missing parameters
                throw_exception(422, missing=missing)
            # returning the result of the wrapped function after manipulation
            return function(*args, **kwargs)
        # returns the replaced function
        return decorated_function
    # returns the real decorator
    return decorator

```

Fig. 4.9: *Decorator* per la richiesta di parametri

```python
def needs_authentication(function):
    @wraps(function)
    def decorated_function(*args, **kwargs):
        user = getUserFromRequest()
        if not user:
            # if here, authentication failed
            throw_exception(401)
        # authenticated with success
        # saving user inside g
        setattr(g, "user", user)
        return function(*args, **kwargs)
    return decorated_function

```

Fig. 4.10: *Decorator* per la richiesta di autenticazione

Ogni modulo contiene inoltre un file *models.py*, dove vengono definite le classi atte alla creazione delle tabelle attraverso l'ORM *sqlalchemy* (Figura 4.12). I modelli del

database derivano (direttamente o indirettamente) da `db.Model`, e hanno definite le colonne come istanze della classe `Column`.

In alcuni moduli, infine, sono presenti un file *utils.py* e *queries.py* per favorire la pulizia del codice e calmierare la complessità del singolo file.

```
1 def getStudent(subject_id, user_id):
2     return Student.query.filter(and_(Student.user_id == user_id,\
3                                      Student.subject_id == subject_id)).first()
4
```

Fig. 4.11: Esempio di query con *sqlalchemy*

I moduli del *web service* sono:

- **ac/base**: all'interno di questo modulo vengono definiti i modelli di base, ovvero quelle classi astratte poi ereditate dalle diverse tabelle degli altri moduli che necessitano di quei particolari attributi. La classe `CommonPK` definisce il contatore che funge da chiave primaria nella quasi totalità delle tabelle (Figura 4.13). La classe `Base` (Figura 4.12), invece, definisce i campi `createdAt` e `updatedAt`, entrambi vengono popolati in automatico e il secondo viene modificato a ogni modifica alla tabella. Inoltre al suo interno è definito la proprietà `json`, per convertire una riga in formato JSON, e `__tablename__`, un `declared_attr` (ovvero un attributo che viene ridichiarato in ogni sottoclasse), che automatizza e standardizza la denominazione delle tabelle. L'unico controller è finalizzato alla comunicazione delle informazioni generali e si trova sul percorso *ip:port/agileclassroom/*.

```python
class Base(db.Model):
    # abstract model
    __abstract__ = True

    # to declare tablename
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

    createdAt = Column(DateTime, default = db.func.current_timestamp())
    updatedAt = Column(DateTime, default = db.func.current_timestamp(),\
                       onupdate = db.func.current_timestamp())

    # getter of the attribute json, so it can be built at runtime
    @property
    def json(self):
        return dict((column.name, getattr(self, column.name))\
                    for column in self.__table__.columns\
                    if hasattr(self, column.name))

```

Fig. 4.12: Modello *Base*

```python
class CommonPK(db.Model):
    # abstract model
    __abstract__ = True

    id =  Column(BigInteger, primary_key = True)

```

Fig. 4.13: Modello *CommonPk*

- **ac/auth**: in questo modulo vengono definiti i modelli `User` e `Keychain`, comprensivi di alcuni metodi utili alla registrazione e all'autenticazione con il sistema *token-based*.
  Inoltre sono definiti i seguenti *endpoint* (*ip:port/agileclassroom/auth*):
  - *login* per il login.
  - *register* per la registrazione.
- **ac/subject**: qui vengono definiti il modello `Subject` e gli endpoint relativi alla materia (*ip:port/agileclassroom/subject/*):
  - *create* per la creazione di una nuova materia.
  - *completion/update* per segnalare come completata una materia.
  - *description/update* per modificare la descrizione di una materia.
  - *name/update* per la modifica del nome.

  - *delete* per la rimozione.
  - *active* per la ricezione delle materie ancora non completate.
  - *completed* per la ricezione delle materie completate.

- **ac/student**: in questo modulo vengono definiti il modello `Student` e gli *endpoint* (*ip:port/agileclassroom/student/*):
  - *add* per l'aggiunta di uno studente ad una classe.
  - *remove* per la rimozione di uno studente da una classe.
  - *left* per l'abbandono di una classe.
  - *get* per la ricezione dell'elenco della classe.

- **ac/module**: qui sono presenti il modulo `Module` e gli *endpoint* (*ip:port/agileclassroom/module/*):
  - *create* per la creazione di un nuovo modulo didattico.
  - *description/update* per la modifica della descrizione.
  - *name/update* per la modifica del nome.
  - *delete* per la rimozione di un modulo didattico.
  - *get* per la ricezione di tutti i moduli didattici di una materia.

- **ac/unit**: contiene la definizione di `Unit` e `Position`

  Sono poi descritti gli *endpoint* relativi a `Unit` (*ip:port/agileclassroom/unit/*):
  - *create* per la creazione di un'unità.
  - *description/update* per la modifica della descrizione.
  - *name/update* per la modifica del nome.
  - *delete* per la cancellazione.
  - *get* per la ricezione di tutte le unità di un particolare modulo.

  E quelli riferiti a `Position` (*ip:port/agileclassroom/position/*):
  - *move* per lo spostamento di un'unità.
  - *student/get* per la ricezione della propria *Scrum Board* (a cura dello studente).

  - *teacher/get* per la ricezione della situazione dei propri studenti per una particolare unità (a cura dell'insegnante).

- **ac/exercise**: modulo contenente i modelli `Exercise`, `TrueFalseAnswer`, `MultipleChoiceAnswer`, `Attempt`, `Proposed` e gli `endpoint` (*ip:port/agileclassroom/exercise/*):

  - *exercise-number* per impostare il numero di esercizi di un'unità.
  - *error-margin* per impostare il margine di errori di un'unità.
  - *create* per creare un nuovo esercizio.
  - *type/update* per modificare il tipo di esercizio.
  - *description/update* per modificare la descrizione dell'esercizio.
  - *true-false/answer* per selezionare la risposta per esercizi vero o falso.
  - *multiple-choice/answer* per selezionare la risposta corretta in un esercizio a scelta multipla.
  - *multiple-choice/new* per aggiungere un'opzione in un esercizio a scelta multipla.
  - *get* per prendere gli esercizi di un'unità.
  - *get-exercise* per prendere i dati di un particolare esercizio.
  - *delete* per cancellare un esercizio.

  Sono inoltre definiti gli endpoint per i tentativi degli studenti di superare la fase di test (*ip:port/agileclassroom/attempt/*):

  - *completion/submit* per rispondere ad un esercizio a completamento.
  - *true-false/submit* per rispondere ad un esercizio vero o falso.
  - *multiple-choice/submit* per rispondere ad un esercizio a scelta multipla.
  - *get* per ricevere la lista dei tentativi effettuati per una particolare unità da un particolare studente.
  - *get-attempt* per ricevere i risultati di un particolare tentativo.

- **ac/security**: qui sono presenti i *decorators* utili alla sicurezza, trattati nel capitolo 8.

4.5 Test Driven Development

4.5.1 Introduzione

Il TDD o *Test Driven Development* è una tecnica che prevede lo sviluppo "guidato" dai test. È un modello di produzione del software in cui la definizione dei test automatici viene effettuata prima dell'applicativo che deve essere sottoposto ai test. Inoltre il software applicativo deve essere orientato esclusivamente a passare quegli stessi test. In questo modo si riducono i tempi di consegna orientando lo sviluppo al solo raggiungimento della commessa.

Il TDD prevede la ripetizione di un breve ciclo di sviluppo composto da tre fasi (*ciclo* TDD, figura 4.14):

1. *Fase rossa*, in cui vengono scritti i test per la nuova funzionalità (che avrà esito completamente negativo, da qui il nome).

2. *Fase verde*, in cui la nuova funzionalità viene implementata e supera tutti i test relativi, senza intaccare i risultati dei test degli altri moduli, condizione fondamentale per la buona riuscita dello sviluppo di un prodotto software.

3. *Fase grigia* o di *refactoring*, in cui il programmatore corregge il codice dell'applicativo per adeguarlo a determinate esigenze.

(Percival, 2014)

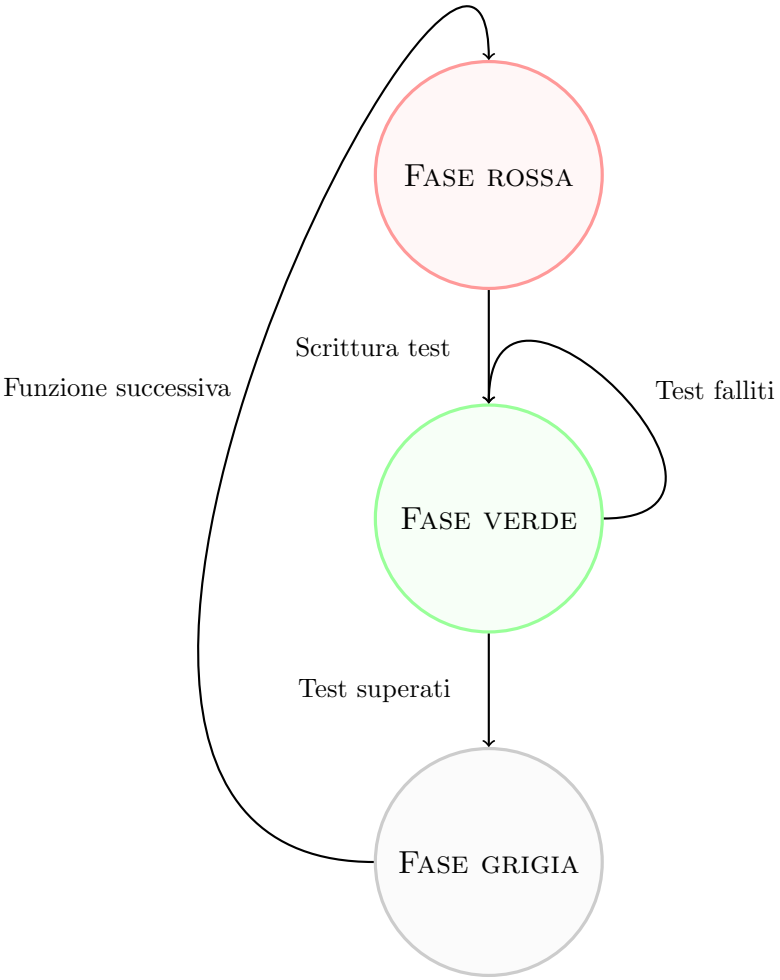


Fig. 4.14: Test Driven Development

4.5.2 Implementazione

L'implementazione si basa sul modulo di Python *unittest*. Questo permette in cooperazione con *Flask* di testare le API dell'applicazione. La strutturazione dei test è la seguente:

1. **test/shared.py**: Questo file contiene la classe base per il test dei moduli, che definisce le operazioni comuni a tutti i test dei diversi moduli e le API per effettuare richieste di tipo GET o POST, semplificando l'accesso ai risultati

in formato JSON (convertendolo in array associativo) e intercettando eventuali errori interni generici (Figura 4.15).

```
import unittest
from ac.exceptions import throw_exception
from flask import jsonify, json
from test.auth.api import register

class ACTestCase(unittest.TestCase):
    def setUp(self):
        # Creating testing application
        from ac import app, db
        self.app = app.test_client()
        db.create_all()
        self.db = db

        # token key
        from ac.auth.utils import TOKEN_KEY
        self.token_key = TOKEN_KEY

        # create user to test some api
        self.username = "username"
        self.password = "password"
        self.email = "email@example.com"
        registration_result = register(self, self.username,\
                                       self.email, self.password)
        self.user = registration_result["json"].get("user")
        self.token = registration_result["json"].get("token")

    def tearDown(self):
        # cleaning db
        self.db.drop_all()

    def get(self, path, token = None, data = None):
        # send get request
        response = self.app.get(path, headers = { self.token_key: token })
        # intercepting internal server error
        if response.status_code == 500:
            print "Internal server error"
            assert False
        # return response
        return { "status_code": response.status_code,\
                 "json": json.loads(response.data) }

    def post(self, path, token = None, data = None):
        # send post request
        response = self.app.post(path, data = data,
                                 headers = { self.token_key: token })
        # intercepting internal server error
        if response.status_code == 500:
            print "Internal server error"
            assert False
        # return response
        return { "status_code": response.status_code,\
                 "json": json.loads(response.data) }

```

Fig. 4.15: Array associativo per la scelta della classe di configurazione appropriata

Ogni modulo di test (parallelo, ovvero identificato dallo stesso nome, ai moduli applicativi) ha due file (oltre a `__init__.py`): `api.py`, che contiene delle funzioni

utili a eseguire richieste (sfruttando i metodi `get` e `post` di `ACTestCase`) e a ritornare la risposta (Figura 4.16), e *tests.py*, che contiene le classi figlie di `ACTestCase` con la definizione dei test (metodi che iniziano con `test_`, figura 4.17).

```
1  def login(self, username, password):
2      return self.post("/auth/login",\
3                       data = {"username": username, "password": password})
4
5  def register(self, username, email, password):
6      return self.post("/auth/register",\
7                       data = {"username": username, "email": email, "password":
       password})
8
9  def get_current_user(self, token):
10     return self.get("/auth/current_user", token = token)
11
```

Fig. 4.16: Esempio di api utilizzate nei test per il modulo *Auth*

```python
from test.shared import ACTestCase
from api import register, login, get_current_user

class AuthTestCase(ACTestCase):
    def test_register(self):
        # data
        # 1. email
        r_email = "test.register@gmail.com"
        w_email = "te.register@gmail.com"
        # 2. username
        r_username = "testregister"
        w_username = "teregister"
        # 3. password
        r_password = "password"


        print "Auth - Registration: Success"
        result = register(self, r_username, r_email, r_password)
        # print "Ok: 200"
        assert result["status_code"] == 200
        # print "Token received"
        assert result["json"].get("token")
        # print "User received"
        assert result["json"].get("user")

        print "Auth - Registration: Existing user --> repeated username"
        result = register(self, r_username, w_email, r_password)
        # print "Error: 403"
        assert result["status_code"] == 403
        # print "Error: repeated username"
        assert result["json"].get("repeated").get("username") == r_username

        print "Auth - Registration: Existing user --> repeated email"
        result = register(self, w_username, r_email, r_password)
        # print "Error: 403"
        assert result["status_code"] == 403
        # print "Error: repeated email"
        assert result["json"].get("repeated").get("email") == r_email

```

Fig. 4.17: Esempio di alcuni test per il modulo *Auth*

Come si può osservare, i test si basano sul comando `assert`, che genera un output solo in caso di condizione falsa, pertanto permettono di scoprire la presenza di anomalie.

2. **test/pipeline.py**: Tutte le classi figlie di `ACTestCase`, che implementano i test per i vari moduli, vengono importate in questo file, che rappresenta l'ordine con cui testare i moduli (Figura 4.18).

```
from base.tests import *
from auth.tests import *
from subject.tests import *
from student.tests import *
from module.tests import *
from exercise.tests import *

```

Fig. 4.18: *test/pipeline.py*

3. **run_tests.py**: Attraverso questo file si possono avviare i file, una volta importata la *pipeline*, attraverso il modulo unittest (Figura 4.19).

```
# importing the tests needed to execute
from test.pipeline import *

# creating app (testing mode)
import ac
ac.create_app("testing")

print "Starting tests.."
# executing tests
import unittest
unittest.main()

```

Fig. 4.19: Avvio dei test

Avviando i test dopo ogni implementazione incrementale e attraverso il meccanismo degli `assert`, in combinazione con i `print`, è possibile intercettare subito ogni possibile errore e/o mancanza.

Capitolo 5

Applicazione Android

Nota: I dati di esempio usati negli esercizi e nelle descrizioni sono presi dal libro "Futuro Impresa 4" (Barale, Rascioni, & Ricci, 2014).

5.1 Accesso e Registrazione

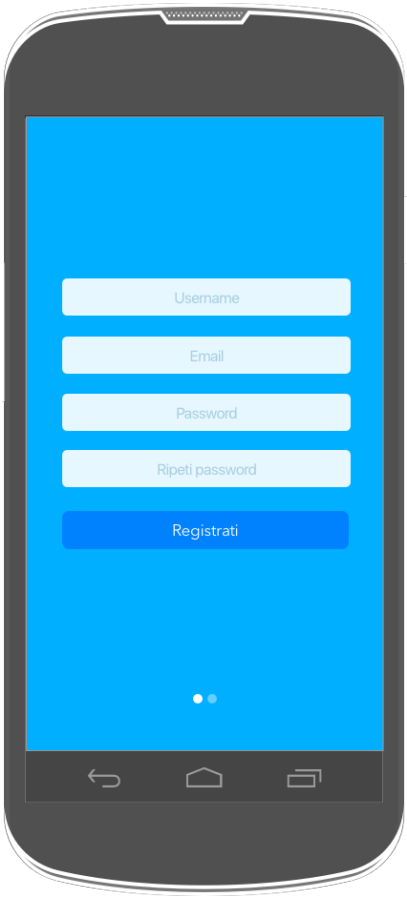


Fig. 5.1: Registrazione

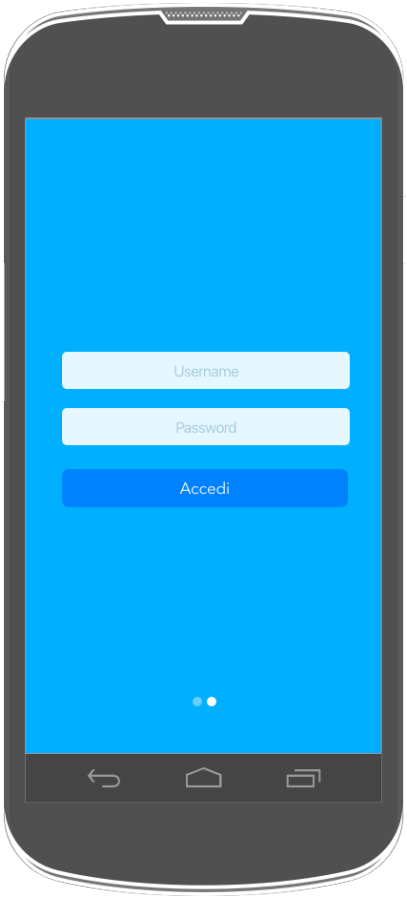


Fig. 5.2: Accesso

All'avvio dell'applicazione viene data all'utente la possibilità di accedere (Figura 5.2) oppure di registrarsi (Figura 5.1), dall'avvio successivo il *login* verrà eseguito in automatico a meno che l'utente non effettui il *logout*. I due `Fragment` sono collegati in un `ViewPager` che permette di passare da una schermata all'altra scorrendo con il dito sullo schermo.

5.2 Home page

Fig. 5.3: Home page - 1

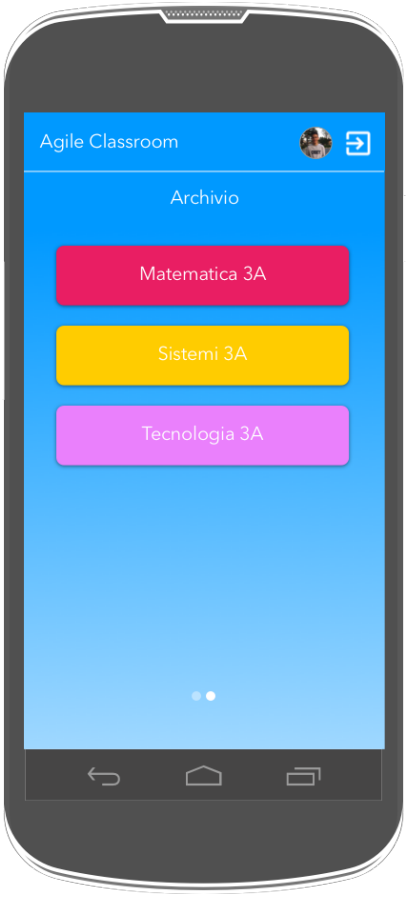


Fig. 5.4: Home page - 2

Nella home page vengono mostrati due `Fragment` in un `ViewPager`: in entrambi è presente una `RecyclerView`, ma mentre nel primo si trovano le materie che lo studente sta ancora studiando (Figura 5.3) nell'altro sono presenti quelle che sono state segnate come già concluse (Figura 5.3). Le materie di cui l'utente è insegnante sono identificate con un segnalibro.

È inoltre possibile, attraverso il `FloatingActionButton` (FAB), creare una nuova materia (di cui l'utente è insegnante). Si considera infatti la possibilità di creare corsi di altro tipo oltre a quelli scolastici, pertanto uno studente può essere anche insegnante.

Infine, cliccando su una materia, è possibile visualizzare le informazioni relative ad essa (Figura 5.5).

È possibile effettuare il logout tramite l'apposita icona nell'`ActionBar`, mentre dall'immagine di profilo si accede alla gestione dello stesso (5.8).

5.3 Materie

Fig. 5.5: Informazioni sulla materia

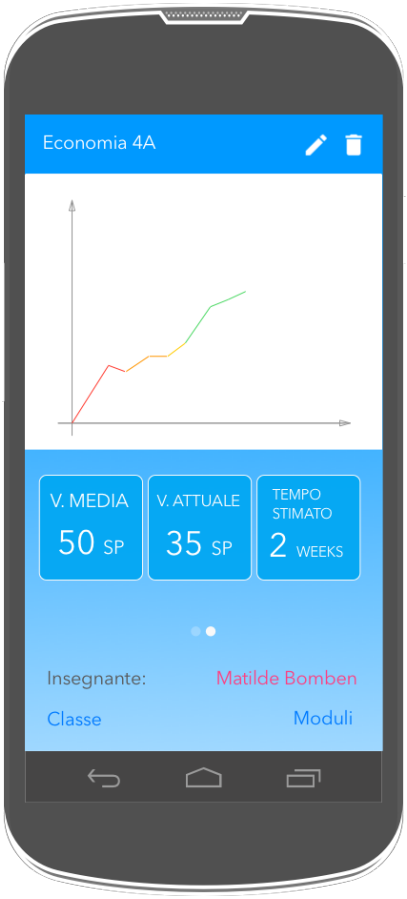


Fig. 5.6: Statistiche della materia

Quando l'utente clicca su una materia vengono mostrate le informazioni sulla stessa (Figura 5.5) e le statistiche sull'avanzamento nello studio (Figura 5.6). Da queste schermate l'insegnante può modificare il nome e la descrizione della materia cliccando sull'icona della matita, e può eliminare la materia cliccando su quella del cestino.

Tutti gli utenti possono scegliere il proprio colore per ogni determinata materia, ma gli studenti non possono modificare altre informazioni riguardo alla stessa, possono solo decidere di abbandonare la classe.

Cliccando sul link *Classe* si accede alla vista degli studenti di quella materia (Figura 5.7), mentre dalla scritta *Moduli* si accede alla lista dei moduli didattici (Figura 5.9).

È possibile passare da un `Fragment` all'altro scrollando con il dito sullo schermo.

5.4 Classe

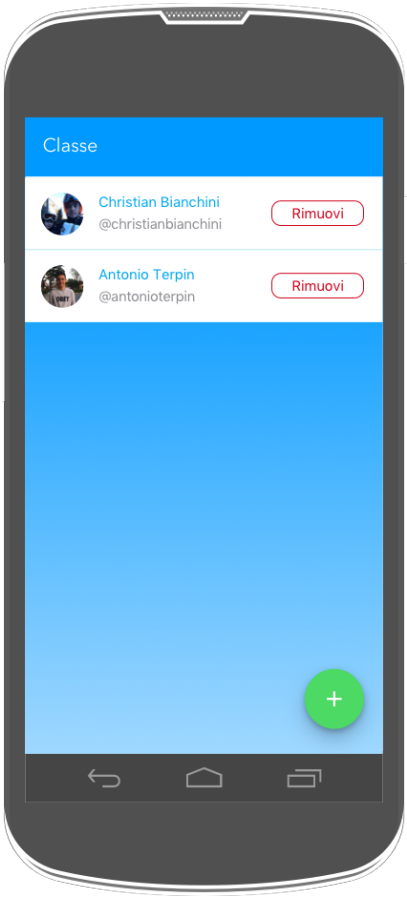


Fig. 5.7: Classe

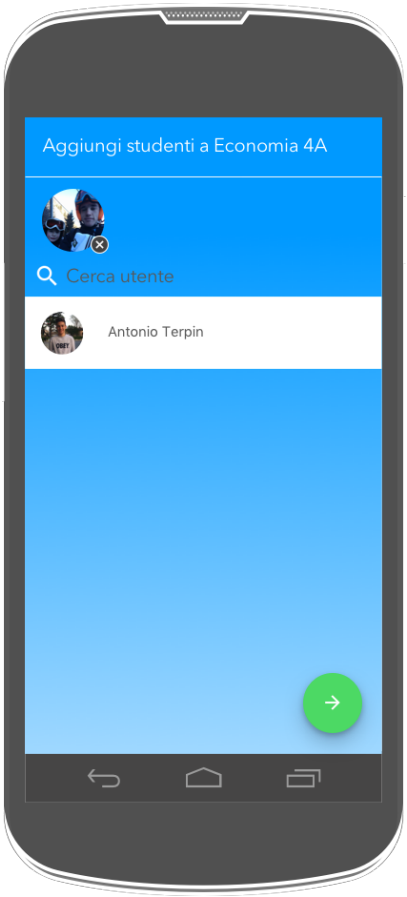


Fig. 5.8: Aggiunta di uno studente

In questa vista (Figura 5.7) sono elencati in una `RecyclerView` i componenti della classe, ovvero tutti gli studenti che sono stati assegnati alla materia; da qui l'insegnante può rimuovere gli alunni oppure, cliccando sul `FloatingActionButton` (FAB), accedere all'*activity* per aggiungerne altri (Figura 5.8). Qui può ricercare degli studenti fra gli utenti non ancora aggiunti, mediante il campo *cerca utente* o dalla `RecyclerView` con l'elenco di questi.

È possibile effettuare una selezione multipla, ottenuta tramite l'aggiunta dell'utente selezionato in una `ListView` orizzontale, posta sopra il campo di ricerca. Cliccando sul `FloatingActionButton` (FAB) in basso a destra è infine possibile confermare l'operazione.

5.5 Moduli didattici

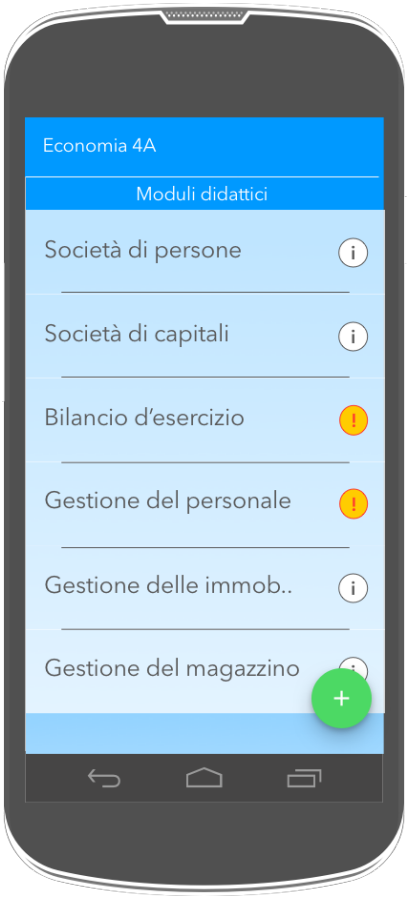


Fig. 5.9: Moduli didattici

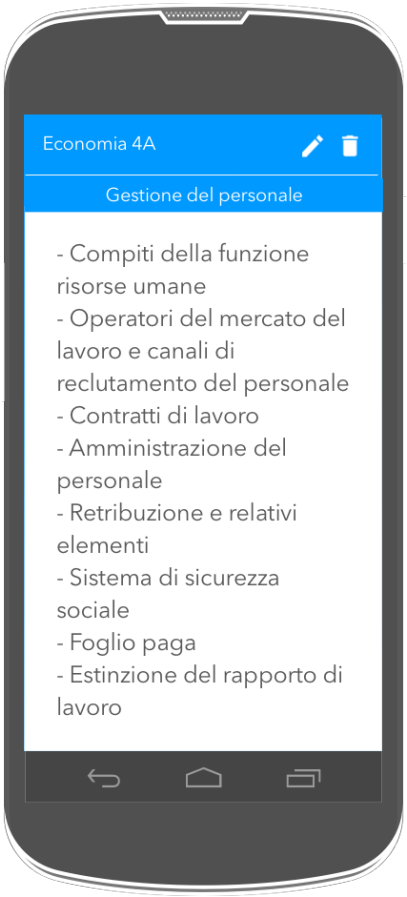


Fig. 5.10: Informazioni sul modulo

Quando l'utente clicca sulla scritta *Moduli* nella vista della materia (Figure 5.5 e 5.6) gli viene mostrata la lista dei moduli didattici che compongono la materia stessa in una `RecyclerView` (Figura 5.9).

L'insegnante ne può aggiungere di nuovi utilizzando il `FloatingActionButton` (FAB) oppure accedere alla lista delle unità didattiche (Figura 5.11) che compongono un modulo, premendo su di esso. Lo studente, invece, quando clicca su uno di essi, accede alla *Scrum Board* relativa al modulo selezionato (Figura 5.16).

Grazie al bottone di informazione si accede alla vista del modulo (Figura 5.10) da cui l'insegnante può modificare il nome e la descrizione dello stesso, mentre lo studente vi accede in sola lettura.

Il bottone informazione può essere sostituito da quello di *alert* (punto esclamativo rosso su arancione) nel caso in cui vi siano problemi in quel particolare modulo.

5.6 Unità didattiche

5.6.1 Lato insegnante

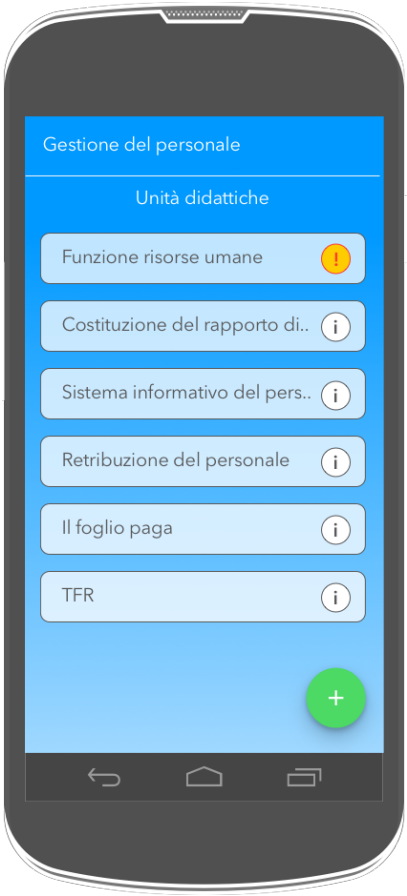


Fig. 5.11: Unità didattiche

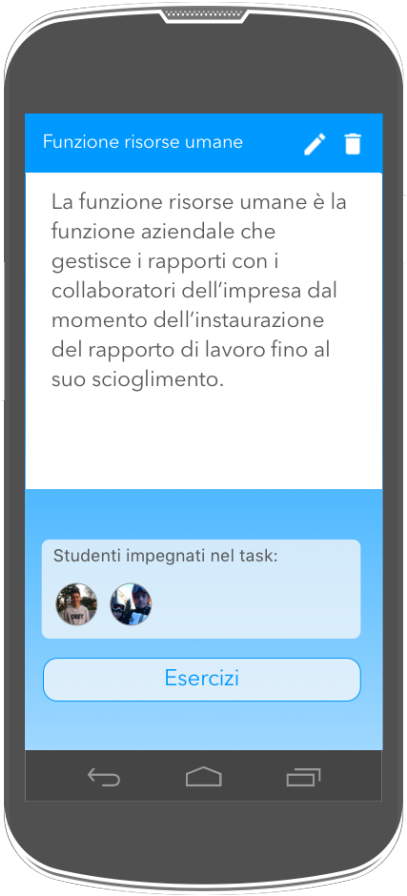


Fig. 5.12: Informazioni sull'unità

La vista delle unità didattiche (Figura 5.11) mostra al docente l'insieme degli argomenti che compongono un modulo in una `RecyclerView`. L'insegnante può aggiungere altre unità cliccando sul `FloatingActionButton` (FAB) oppure visualizzare le informazioni di quelle esistenti (Figura 5.12) premendo su di esse. Da quest'ultima sezione è possibile vedere gli studenti impegnati nel task in una `ListView` orizzontale e accedere alla gestione degli esercizi di quella particolare unità.

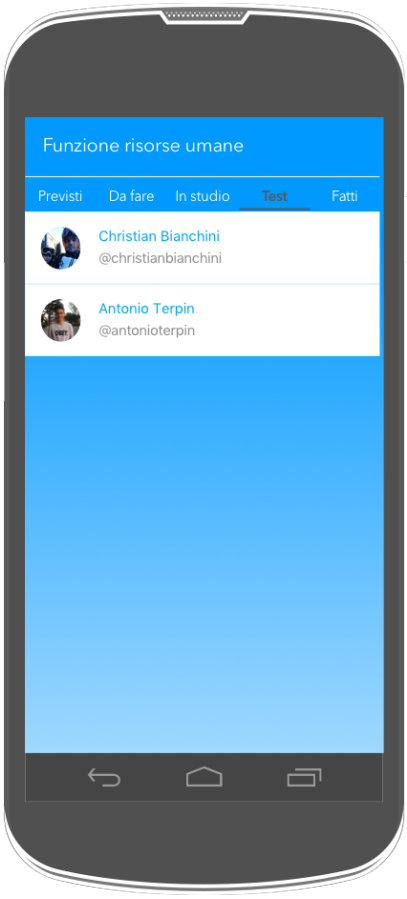


Fig. 5.13: Situazione della classe

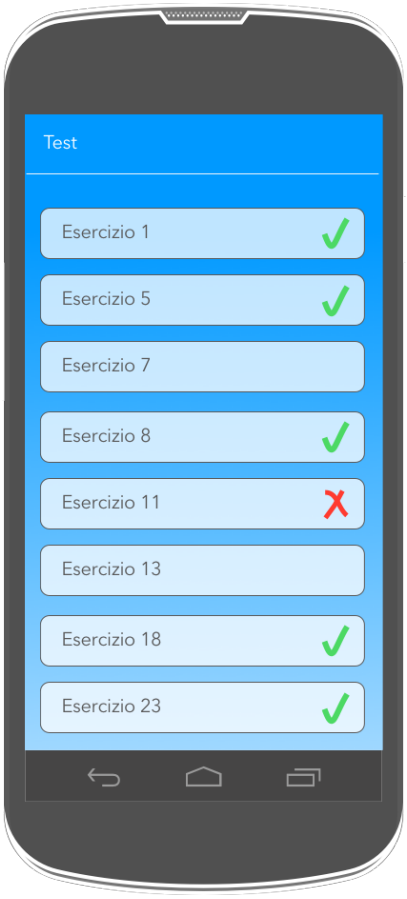


Fig. 5.14: Test dello studente

Cliccando sul pulsante di informazione si accede alla *Scrum Board* rappresentante la situazione della classe in quel singolo argomento (Figura 5.13) da cui si possono visualizzare i risultati degli esercizi svolti da uno specifico studente (Figura 5.14), selezionando la situazione dello stesso.

La spunta verde indica la correttezza nell'esecuzione dell'esercizio, mentre quella rossa segnala un errore. Se non è presente alcun simbolo significa che l'esercizio non è ancora stato completato.

5.6.2 Lato studente

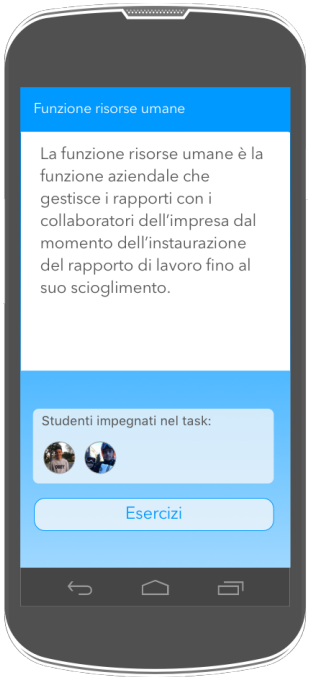


Fig. 5.15: Descrizione dell'unità

Fig. 5.16: Scrum board

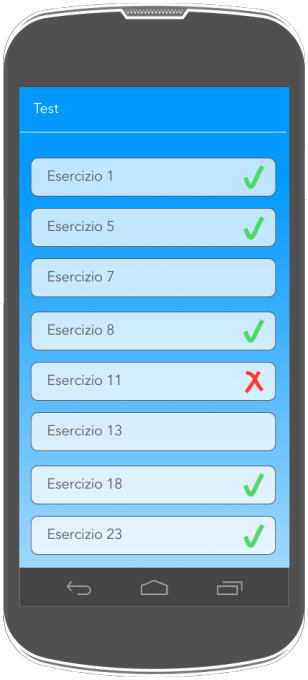


Fig. 5.17: Informazioni sul test

Accedendo a un modulo, lo studente visualizza la *Scrum Board* (Figura 5.16) con il proprio avanzamento nelle varie unità. La `ScrumBoard` è ottenuta tramite un `ViewPager` e delle `ListView` dalle quali è possibile trascinare e spostare nelle varie pagine la singola unità.

È possibile accedere alle informazioni del singolo argomento (Figura 5.15) e allo storico dei test effettuati dallo studente (Figura 5.17).

Una volta che l'unità si trova in fase di test, la stessa è bloccata fino al completamento di questo, e solo in caso di completamento viene mossa automaticamente nella sezione *fatti*, altrimenti ritorna a quella *in studio*.

5.7 Esercizi

5.7.1 Lato insegnante

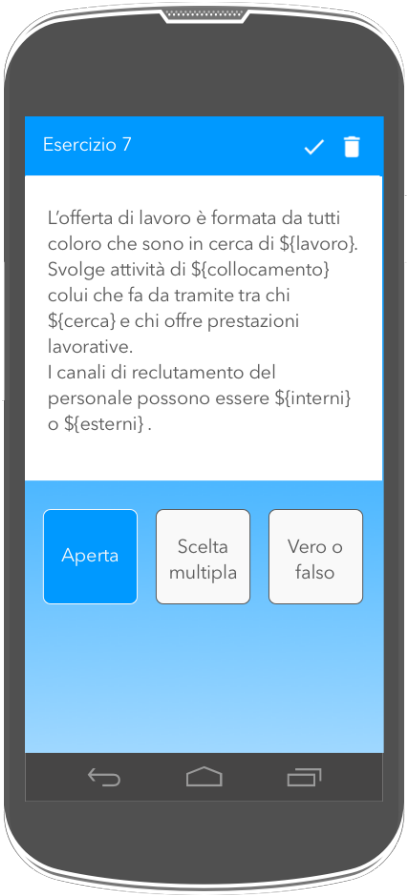


Fig. 5.18: Esercizio a completamento

Fig. 5.19: Vero o falso

L'applicazione rende possibile l'inserimento di esercizi di diverso tipo, in base all'opzione scelta tra i tre bottoni (*Aperta*, *Scelta multipla* o *Vero o falso*).

Gli esercizi a completamento (*Aperta*) sono definiti con la sola descrizione. Le parti di frase da inserire sono nella forma `${testo}` (Figura 5.18). Le domande vere o false sono invece caratterizzati da una risposta (quella corretta è blu).

Fig. 5.20: Risposta multipla (domanda)

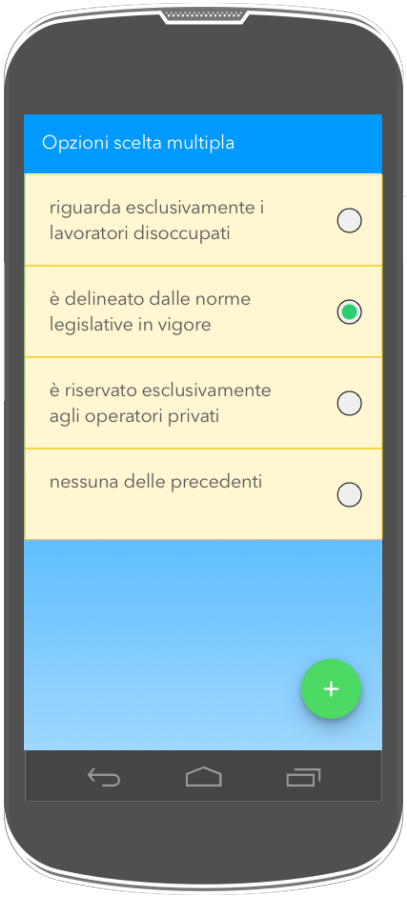


Fig. 5.21: Risposta multipla (risposta)

Gli esercizi a scelta multipla sono caratterizzati da una domanda (Figura 5.20) e da n risposte, gestibili nella schermata raggiunta attraverso il bottone *Gestisci soluzione*. Le risposte sono inserite in una `RecyclerView` e quella corretta è quella con il `radio button` selezionato. È possibile aggiungere un'ulteriore risposta con il `FloatingActionButton` (FAB) (Figura 5.21).

È possibile inoltre scegliere il numero di quesiti da sottoporre agli studenti e il numero massimo di errori che l'allievo può commettere affinchè il test sia considerato come completato.

5.7.2 Lato studente

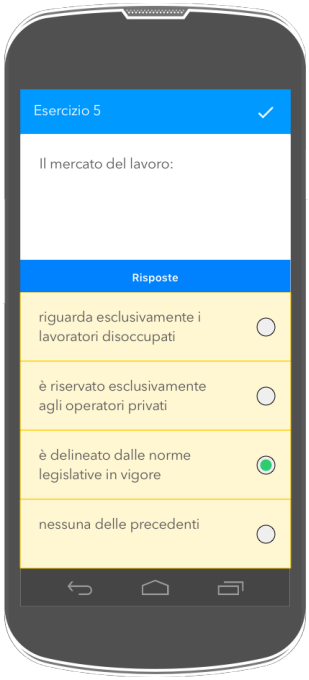


Fig. 5.22: Esercizio a risposta multipla

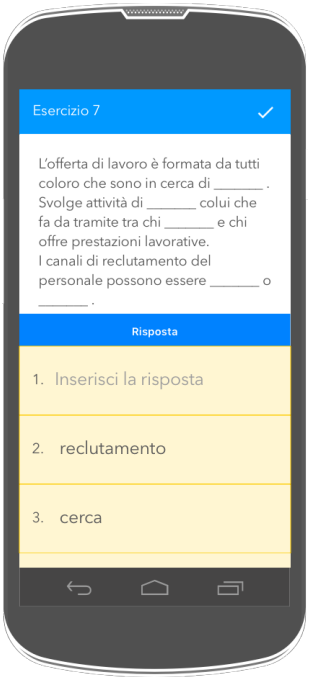


Fig. 5.23: Esercizio a completamento

Fig. 5.24: Esercizio vero o falso

Lato studente è possibile accedere al test solamente una volta che l'unità sia stata posta in fase di test. L'allievo, a questo punto, dovrà rispondere a un numero di quesiti scelto dall'insegnante e, se il numero di errori commessi è minore o uguale al limite impostato dal docente, il test viene superato e l'unità viene segnalata come completata, altrimenti viene posta nuovamente nella sezione precedente.

Selezionando la spunta nella `ActionBar` si può sottoporre l'esercizio a correzione (Figura 5.22, 5.23 e 5.24).

5.8 Profilo

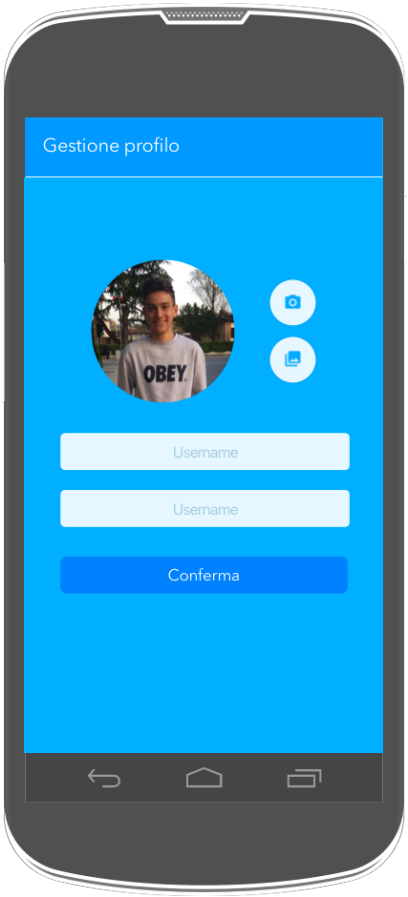


Fig. 5.25: Impostazioni del profilo

È possibile accedere alle impostazioni del proprio profilo (Figura 5.25) premendo sull'immagine presente nell'`ActionBar` presente nella *Home Page*. Da qui si possono modificare le proprie informazioni e la propria immagine di profilo.

Capitolo 6

Comunicazione Client-Server

6.1 Lato server

La comunicazione tra *Client* e *Server* avviene utilizzando come formato di interscambio dati JSON (Appendice D), di cui è possibile vedere un esempio alla figura 6.1.

```
message = {
    "name"              :  "AgileClassroom Webservice",
    "version"           :  1.0,
    "webservice type"   :  "REST",
    "last_run"          :  "Thu, 01 Jun 2017 10:39:05 GMT"
}

```

Fig. 6.1: Esempio di messaggio

Per quanto concerne la trasmissione, il webservice si appoggia alla funzione per la gestione delle risposte in JSON offerta da *Flask* (Capitolo 4): jsonify.

La figura 6.2 da come risultato la figura 6.1.

```
from flask import Blueprint, jsonify
from datetime import datetime
import pytz
from ac import app

base = Blueprint("base", __name__, \
                 url_prefix= "/agile-classroom")

@base.route("/", methods = ["GET"])
def welcome():
    return jsonify(
        name = app.config["NAME"], \
        version = app.config["VERSION"], \
        webservice_type = app.config["TYPE"], \
        last_run = app.config["LAST_RUN"])

```

Fig. 6.2: Esempio di utilizzo di jsonify

Infatti gli oggetti in PYTHON sono accessibili come array associativi. Inoltre, per convertire in un oggetto JSON un'istanza di un modello del database (*model*, si veda il capitolo 4, sezione 4.4) si utilizza la proprietà `json` della classe `Base`, ereditata da tutti i modelli del database (Figura 6.3).

```
@property
def json(self):
    return dict((column.name, getattr(self, column.name)) \
        for column in self.__table__.columns \
        if hasattr(self, column.name))

```

Fig. 6.3: Proprietà *json* di *Base*

Una proprietà in PYTHON rappresenta un metodo di accesso ad un attributo, quello che in altri linguaggi viene definito come *getter*. In particolare, il risultato del metodo è dato dalla costruzione di un array associativo (*dictionary*) ottenuto tramite la funzione di PYTHON `dict` e un'espressione generica (*lambda expression*) che itera per ogni colonna della riga e estrae una coppia chiave valore se il valore esiste (vengono quindi rimosse le colonne in cui la riga in questione ha valore nullo): nome della colonna - valore della riga in quella colonna (cella).

La proprietà `json` viene combinata con la funzione `jsonify` dal webservice per ritornare qualunque tipo di risultato (Figura 6.4).

```
return jsonify(units = [unit.json \
        for unit in getUnitsAPI(g.query.get("module_id"))])

```

Fig. 6.4: Jsonify utilizzato con un'entità del database

Nella figura 6.4 si può notare come la *lambda expression* iteri tra tutte le righe ottenute dalla funzione `getUnitsAPI` e come le inserisca in un array come oggetti JSON tramite la proprietà `json`. Un risultato plausibile da questa istruzione è quello proposto alla figura 6.5

```
units = [{
        "id"          :  1,
        "createdAt"   :  "2017-05-24 17:25:11.875425",
        "updatedAt"   :  "2017-05-24 17:25:11.875425",
        "name"        :  "Il foglio paga",
        "description" :  "Documento elaborato [..]",
        "numberOfEx"  :  13,
        "errorMargin" :  1
},{
        "id"          :  2,
        "createdAt"   :  "2017-05-24 17:25:11.875425",
        "updatedAt"   :  "2017-06-01 21:12:14.098345",
        "name"        :  "Funzione risorse umane",
        "description" :  "La funzione risorse [..]",
        "numberOfEx"  :  10,
        "errorMargin" :  2
}]

```

Fig. 6.5: Risultato figura 6.4

Per quanto riguarda la ricezione, questa viene gestita intrinsecamente, con architettura REST (Appendice C, sezione C.3), dal webservice attraverso il framework *Flask* (si veda il capitolo 4).

6.2 Lato client

Per permettere all'applicazione di effettuare le richieste verso il *web service* è stata usato *Retrofit*, una libreria di Android e Java che serve a gestire le richieste HTTP (e HTTPs) attraverso l'utilizzo delle interfacce di Java.

6.2.1 Interfaccia

Un esempio di un'interfaccia usata per ottenere dei dati dal *web service* è quello descritto in figura 6.6, in cui viene creata l'API per interagire con il modulo che gestisce gli studenti.

```
public interface StudentsAPI {
   @GET("agileclassroom/student/get")
   Call<List<Student>> getStudents(@Query("subject_id") int subjectId);

   @POST("agileclassroom/student/add")
   Call<Student> addStudent(@Query("subject_id") int subjectId, @Query("user_id")
       int userId);

   // Altri metodi del modulo Students
}
```

Fig. 6.6: Esempio di API per scaricare l'elenco dei componenti di una classe

Come si può vedere, *Retrofit* usa le annotazioni per rappresentare il metodo e l'URI della richiesta HTTP: i metodi disponibili come annotazioni sono `@GET`, `@POST`, `@PUT`, `@DELETE` e `@HEAD`, mentre l'URI viene scritto come stringa relativa al percorso base impostato per *Retrofit* (Figura 6.7).

6.2.2 Utilizzo

```
Retrofit retrofit = new Retrofit.Builder()
   .baseUrl("http://path.of.server.srv/")
   .build();

StudentsAPI studentsAPI = retrofit.create(StudentsAPI.class);
studentsAPI.getStudents(1).enqueue(getStudentsCallback);
```

Fig. 6.7: Esempio di utilizzo di un API con Retrofit

Per poter disporre delle funzionalità di *Retrofit* è necessario istanziare un istanza della classe `Retrofit` attraverso l'utilizzo di un oggetto `Retrofit.Builder`, dopodichè si procede a creare un oggetto dell'interfaccia dell'API (Figura 6.7).

Tale oggetto servirà per poter inviare le richieste attraverso i metodi definiti nell'interfaccia, ai quali va assegnato un oggetto `Callback` per poter gestire la risposta.

6.2.3 Callback

Ogni volta che si vuole utilizzare un metodo di una API bisogna associargli un `Callback` (Figura 6.8) in modo da poter eseguire delle azioni una volta che la richiesta sia stata completata.

```
Callback<List<Student>> getStudentsCallback = new Callback<List<Student>>() {
   @Override
   public void onResponse(Call<List<Student>> call, Response<List<Student>>
       response) {
      if (response.isSuccessful()) {
         // Operazioni con la risposta
      }

   @Override
   public void onFailure(Call<List<Student>> call, Throwable t) {
      t.printStackTrace();
   }
};
```

Fig. 6.8: Esempio di creazione di un Callback

Capitolo 7

Sincronizzazione dei dati

7.1 Analisi della problematica

Dal momento che ogni applicazione client mantiene in locale una copia di un sottoinsieme del database, risulta necessario tenere sicronizzata la rete di *database distribuiti* che si viene a creare in modo da permettere all'utente di lavorare su una copia sempre aggiornata della base di dati.

La soluzione più semplice risulta essere quella di dare la possibilità all'utente di poter utilizzare l'applicazione anche offline ma in sola lettura: in questo caso sarebbe quindi possibile visualizzare i dati senza però modificarli (non sarebbe quindi possibile muovere gli elementi all'interno della *Scrum Board* oppure svolgere gli esercizi o ancora creare nuove materie, moduli o unità).

Una versione più avanzata sarebbe invece quella di consentire di modificare i dati anche senza connessione a Internet e, una volta che si disponga di un accesso alla rete, sincronizzare il database risolvendo in automatico o con la conferma dell'utente eventuali *conflitti*.

7.2 Soluzione

La prima soluzione può essere implementata effettuando una sincronizzazione completa oppure attraverso un aggiornamento incrementale, mentre la seconda necessita di utilizzare una tecnica di sincronizzazione per i *database distribuiti*.

7.2.1 Sincronizzazione completa

La sincronizzazione completa prevede di ottenere completamente tutti i dati relativi alla schermata che si sta visualizzando in quel momento, per esempio nel caso si acceda

alla *Home Page* verrebbe scaricata la lista completa delle materie di cui l'utente è insegnante o studente.

Il vantaggio di utilizzare questo metodo è che si dispone ogni volta della copia esatta dei dati contenuti nel database centrale, di contro si ha il rischio di dover trasmettere una grossa mole di dati anche nel caso di un piccolo aggiornamento.

7.2.2 Sincronizzazione incrementale

La sincronizzazione incrementale, invece, consente di ridurre questa quantità facendo in modo di richiedere solo i dati che sono stati modificati dall'ultimo aggiornamento. Sempre utilizzando l'esempio della *Home Page*, l'applicazione richiederà la lista delle materie inviando anche il *timestamp* dell'ultimo aggiornamento, e a quel punto il server provvederà a restituire il sottoinsieme delle materie che è stato modificato dopo quel *timestamp*.

Lo svantaggio di questa tecnica è il fatto di dover tener traccia anche degli elementi che vengono cancellati.

7.2.3 Sincronizzazione offline

La sincronizzazione nel caso dei *database distribuiti*, in questo caso chiamata *sincronizzazione offline*, è la soluzione più professionale ma anche quella di più difficile implementazione. L'obiettivo di questa tecnica è quello di permettere all'utente di utilizzare tutte le funzionalità anche quando non è collegato ad internet.

L'idea di base è quella di utilizzare dei meccanismi di *version control* al fine di tenere traccia di tutte le modifiche, *online* e *offline*, e poi risolvere in maniera automatica i *conflitti* che ne possono scaturire, o in caso di difficoltà richiedere l'intervento dell'utente.

Non si dispone di ulteriori informazioni su quest'ultima opzione poichè la si prevede come sviluppo futuro.

Capitolo 8

Sicurezza

L'autenticazione di un utente presso il *web service* è *token-based*, ovvero si basa una stringa pseudo casuale che l'utente invia come header nelle richieste che effettua (Figura 8.1).

```
{
  "ac-session-token": "eyJhbGciOiJIUzI1NiJ9[..]grqTI"
}

```

Fig. 8.1: Autenticazione *token-based*

Questo token viene utilizzato per ovviare alla condizione di protocollo *stateless* di HTTP, per cui operando in questo modo è possibile mantenere traccia della sessione. Lo si ottiene al login o alla registrazione, operazione durante la quale l'utente dimostra con la sua password la propria identità.

Il sistema di autorizzazione per l'accesso alle risorse si basa su un sistema di *decorators* presente nel modulo *ac/security* del *web service*. Ogni risorsa potrà non essere accessibile ad un sottoinsieme di utenti, essere accessibile ai soli insegnanti, ai soli studenti o a entrambe le categorie di utenti (Figura 8.2).

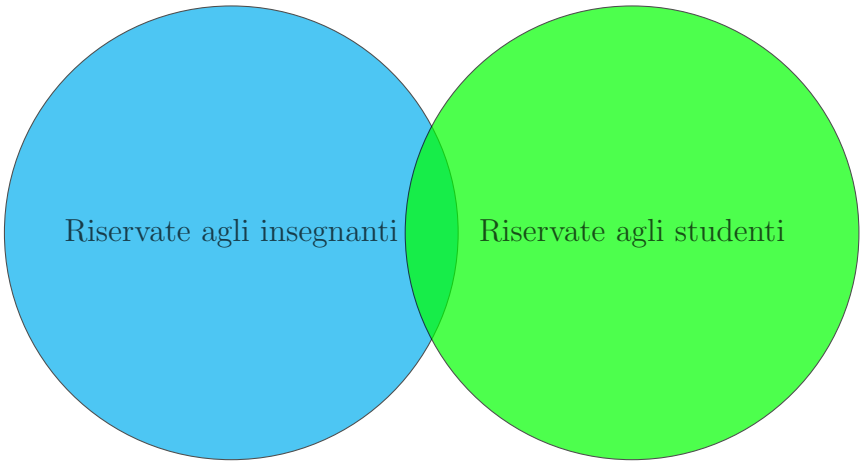


Fig. 8.2: Risorse della piattaforma

Pertanto si è resa necessaria la costituzione dei seguenti *decorators*:

- `techer_zone` (Figura 8.3), che cerca la materia a cui è riferita l'API e di conseguenza concede o no l'accesso, dopo aver consultato il DB.

```
def teacher_zone(method):
    def decorator(function):
        @wraps(function)
        def decorated_function(*args, **kwargs):
            container = getattr(g, outputKeyForMethod(method))
            # get subject
            subject = getSubject(container["subject_id"])
            if not subject or subject.teacher_id != g.user.id:
                # unauthorized
                throw_exception(403, error = "Non sei insegnante di questa materia!")
            checkUnitModuleInSubject(container, subject)
            return function(*args, **kwargs)
        return decorated_function
    return decorator

def checkUnitModuleInSubject(container, subject):
    # module.subject != subject?
    if "module_id" in container.keys()\
       and not isModuleInSubject(container["module_id"], subject.id):
        # unauthorized
        throw_exception(403, error = "Non puoi accedere a questo modulo!")
    # unit.subject != subject?
    if "unit_id" in container.keys()\
       and not isUnitInSubject(container["unit_id"], subject.id):
        # unauthorized
        throw_exception(403, error = "Non puoi accedere a questa unita'!")

```

Fig. 8.3: Riservato agli insegnanti

Questo *decorator* controlla la presenza di un `subject_id`, ovvero di una materia di riferimento per filtrare la richiesta, e controlla che l'utente sia effettivamente

insegnante di quella classe. Infine controlla, nel caso in cui si stia operando su moduli o unità didattici, che questi effettivamente appartengano alla materia di cui l'utente è insegnante, per evitare questo tipo di *exploit*.

Si rende presente che `container` è un array associativo contenente i valori della richiesta, ed è un attributo dell'oggetto `g` (`g` rappresenta la *cache* della richiesta e permette di memorizzare i parametri ricevuti; l'indice in `g` cambia in base al metodo della richiesta, da cui la funzione `outputKeyForMethod`, infatti questi parametri sono posizionati in una particolare posizione dal *decorator* `needs_parameters`, visibile al capitolo 4, figura 4.9).

- `student_zone` (Figura 8.4), che permette o no l'accesso e la manipolazione di una risorsa ai soli studenti di una materia.

```
def student_zone(method):
    def decorator(function):
        @wraps(function)
        def decorated_function(*args, **kwargs):
            container = getattr(g, outputKeyForMethod(method))
            # get subject
            subject = getSubject(container["subject_id"])
            if not subject or g.user.id not in\
              [student.user_id for student in getStudents(subject.id)]:
                # unauthorized
                throw_exception(403, error = "Non sei studente di questa materia!")
            checkUnitModuleInSubject(container, subject)
            # authorized
            return function(*args, **kwargs)
        return decorated_function
    return decorator

```

Fig. 8.4: Riservato agli studenti

- `subject_zone`, che verifica esclusivamente che l'utente sia studente o insegnante della materia (Figura 8.5).

```python
def subject_zone(method):
    def decorator(function):
        @wraps(function)
        def decorated_function(*args, **kwargs):
            container = getattr(g, outputKeyForMethod(method))
            # get subject
            subject = getSubject(container["subject_id"])
            if not subject or not\
              (subject.teacher_id == g.user.id or g.user.id in\
               [student.user_id for student in getStudents(subject.id)]):
                # unauthorized
                throw_exception(403, error = "Non sei studente di questa materia!")
            checkUnitModuleInSubject(container, subject)
            return function(*args, **kwargs)
        return decorated_function
    return decorator

```

Fig. 8.5: Riservato agli utenti della materia

Per quanto concerne la sicurezza della comunicazione, ovvero per garantire riservatezza e integrità, ci si è appoggiati su HTTPs (Appendice F). Per farlo è stato sufficiente modificare il file *run.py* nel modo descritto nella figura 8.6.

```python
# init application
import ac
ac.create_app("development")

# relay on ssl
from ac import app
from OpenSSL import SSL
context = SSL.Context(SSL.SSLv23_METHOD)
context.use_privatekey_file(app.config["SERVER_KEY"])
context.use_certificate_file(app.config["SERVER_CERTIFICATE"])

# running app on secure socket layer
app.run(host = app.config["HOST"], port = app.config["PORT"],\
        debug = app.config["DEBUG"], ssl_context = context)


```

Fig. 8.6: Applicazione avviata su HTTPs

Infine, le difese da *sql injection* sono previste dall'ORM utilizzato (*SQLalchemy*).

Capitolo 9

Altro

9.1 Versionamento

0.1 Applicazione Android, database locale *MySQLi*, unico utente, materie

0.2 Aggiunta moduli, elenco unità

0.3 Multi utente, classi per ogni materia

0.4 *Scrum board* studente

0.5 *Scrum board* insegnante

0.6 Fase di test e esercizi a correzione automatica

0.8 WebService in PYTHON, database *PostreSQL*, comunicazione con HTTP

0.9 Comunicazione con HTTPS

1.0 Layer di sicurezza contro *sql injection* e *blind attacks*

9.2 Ulteriori sviluppi

1.1 Fix errori e bug riscontrati durante l'*alpha release*

1.2 Introduzione sistema di domande-risposte per supportare gli studenti nella ricerca di informazioni riguardo le diverse unità

1.5 Automatizzazione nella creazione di *materie*, *moduli* e *unità* (base comune)

1.9 Applicazione WEB

2.0 Automatizzazione nella creazione di esercizi

3.0 Funzionalità di supporto durante la spiegazione (esercizi da svolgere in classe) e possibilità di caricare materiale di supporto

9.3 Risorse utilizzate

- ATOM per lo sviluppo del *backend* in PYTHON
- POSTMAN per il test delle api del *backend*
- ANDROIDSTUDIO per lo sviluppo dell'applicazione Android
- APACHE per il *deployment* del *web service* su TLS/SSL
- VAGRANT per il *development* del *backend*
- GIT e GITLAB per il software di *version control*

Appendice A

Il tempo, una risorsa preziosa

Alice: Per quanto tempo è per sempre?
Bianconiglio: A volte, solo un secondo.

Lewis Carroll

Il tempo è una dimensione che da sempre ha affascinato l'uomo per l'intrinseco legame con la sua esistenza, in particolare con la morte; «La vita fugge, et non s'arresta una hora», (Petrarca, XIV secolo). Il tempo c'è perchè si esiste?

Mentre nel mondo antico questo aveva una dimensione quantitativa e numerica, tra gli intellettuali che sostennero l'esistenza di un tempo scientifico e di uno della coscienza vi fu Bergson, che enfatizzò l'esistenza di una percezione personale e soprattutto non lineare dello stesso. Per quanto questo già possa sembrare dirompente, la scienza si spinse addirittura oltre. Einstein, con le sue due teorie della relatività, finisce per demolirlo e tutto ciò si ripercuote anche nel modo di scrivere, pensare e agire.

Un esempio significativo è dato dalle avanguardie letterarie; Joyce introduce la tecnica del flusso di coscienza, che evidenzia l'infinita e irrazionale dimensione interiore dell'uomo, nell'*Ulisse*, in cui una singola giornata, il 16 giugno del 1904, viene narrata in 732 pagine, con uno scavo psicologico senza precedenti. Similmente Svevo, incoraggiato proprio da Joyce, compone il suo terzo e celebre romanzo *La coscienza di Zeno*, in cui il fluire della coscienza, attraverso la memoria e la tecnica del monologo interiore, si trova alla base del racconto, confermando così la mancanza di assolutezza della dimensione temporale.

La sfida di definire il tempo è sempre più ardua. È molto più semplice concordare con Einstein e proclamarne la non esistenza. Sostanzialmente, il ventesimo secolo porta al declino il concetto del tempo, per cui non solo «Un'ora, non è solo un'ora, è un vaso colmo di profumi, di suoni, di progetti, di climi.» (Proust, 1913 - 1927), ma è addirittura nulla.

Perchè la realtà e il tempo sono quindi una così tenace illusione? (Einstein, 2000) Se il secondo non esiste in scala microscopica, certamente appare come proprietà emergente della materia/energia (Greco, 2014). Sostanzialmente, anche se la scienza ha dimostrato l'inesistenza del tempo, possiamo affidarci all'intuizione per dare valore a questa parola, senza affannarci troppo a definirla, che è ciò che così intelligentemente Sant'Agostino ha spiegato nelle sue *Confessioni*: «Che cos'è insomma il tempo? Lo so finchè nessuno me lo chiede; non lo so più, se volessi spiegarlo a chi me lo chiede» (Sant'Agostino d'Ippona, 400 ca.).

Questa stessa idea di un tempo come oggetto intuitivo appare nel pensiero di Schopenhauer, che lo identifica come una delle tre forme *a priori*, insieme a *spazio* e *causalità*, che sono gli unici punti comuni nel processo di formazione del mondo ad appannaggio degli uomini. Si ha quindi addirittura un mondo che è inconoscibile e una realtà che viene coperta dal *velo di Maya* o, secondo Marx, dalla *sovrastruttura* della classe dominante.

Per quanto sia arduo comprendere un mondo inafferrabile e, anzi, solo intuibile, secondo l'idea del *varco* di Montale, e nonostante sia complicato concepire qualcosa di così astratto, si può percepire almeno come lo scorrere del tempo, azione che per quanto inesistente ci convince, sia inesorabile e irreversibile, e se ne può pertanto apprezzare l'inestimabile valore.

Appendice B

Scrum

> Non otterrete nulla se vi limitate a parlarne. L'azione è importante.
>
> *W. Edwards Deming*

Il mondo moderno è estremamente veloce, si evolve sempre più rapidamente, e in un ambiente del genere pianificare le proprie azioni è certamente utile, ma lo è di meno seguire ciecamente ciò che si ha pianificato. Non si può più pensare che qualcosa progettato anni prima sia di per certo adatto al contesto attuale. Con il tempo sorgono problemi, idee nuove e soluzioni migliori, che non si possono lasciare in disparte per seguire i numerosi diagrammi di una documentazione scritta a priori. È meglio ispezionare e controllare a brevi intervalli, verificando che ciò che è stato fatto sia ancora quello che si farebbe in quel momento, e riscontrare rapidamente problemi o pareri negativi, per poter riparare agli errori, piuttosto che investire un'infinità di tempo in un'attività di cui non si conosce veramente il valore. *Scrum* (dal termine inglese che indica la mischia del rugby) è la strategia inventata da Jeff Sutherland negli anni '90 per rendere più produttivi i *team* nell'industria del software, ma ora utilizzata dalle più importanti aziende, da Amazon a Google, da Apple a Ferrari, per aumentare la propria produttività (Sutherland, 2016).

Questo approccio propone la realizzazione di un prodotto secondo una filosofia incrementale, aggiungendovi valore di volta in volta e ricevendo *feedback* ravvicinati nel tempo. Si oppone al classico approccio *waterfall* (o a cascata) che prevede una fase di pianificazione totale all'inizio e poi una sola fase di sviluppo e rilascio.

B.1 Principi

* SHU HA RI (守破離), un concetto proveniente dalle arti marziali giapponesi che prevede tre fasi di apprendimento:
  - shu (守, "obbedire", seguire la tradizione), imparare i fondamenti, le tecniche, i proverbi.
  - ha (破, "allontanarsi", rottura con la tradizione), iniziare a sperimentare cose nuove con le conoscenze acquisite.
  - ri (離, "separarsi", trascendenza), non ci sono più tecniche o proverbi, tutti i movimenti sono naturali.
* PDCA (Plan Do Check Act, Pianificare Eseguire Controllare Agire), detto anche *Ciclo di Deming* dal nome dell'ideatore, Deming W. Edwards (Figura B.1), che prevede quattro fasi nello sviluppo e nel controllo di un'attività:
  1. Plan, pianificare quello che si andrà a svolgere nell'**immediato** futuro.
  2. Do, svolgere quanto pianificato.
  3. Check, controllare e analizzare il processo e i relativi risultati alla ricerca di problemi da risolvere con determinate azioni correttive.
  4. Act, agire mettendo in pratica le azioni correttive individuate, ricominciando il ciclo.

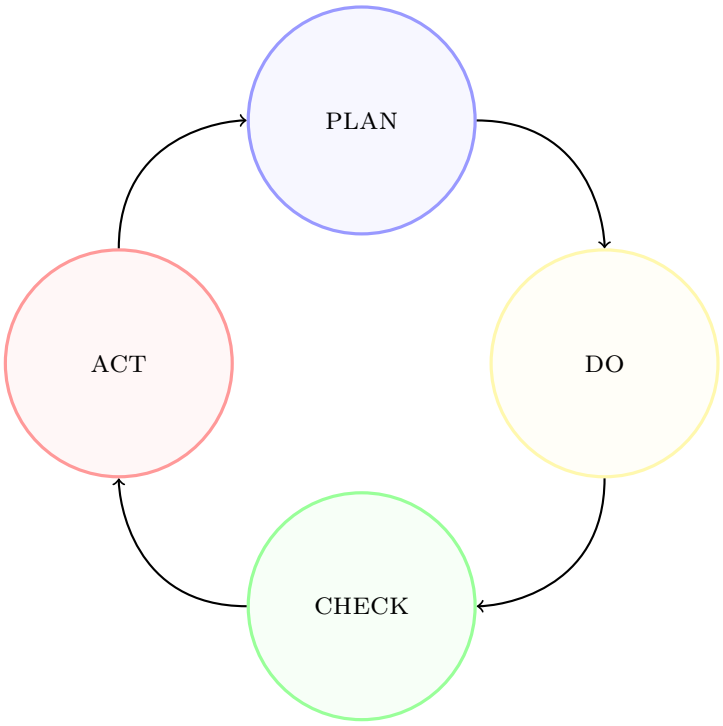


Fig. B.1: Ciclo di Deming (PDCA)

* Gestione LEAN, mirata alla rimozione degli sprechi, ricercabili non solo nello spreco di risorse ma anche nelle attività umane:
  - Lo spesso osannato *Multitasking*, lo svolgere più azioni contemporaneamente, è nella pratica prerogativa solo del 2% della popolazione mondiale, per gli altri si riconduce ad un crollo della produttività (sotto il 40%), secondo uno studio dell'APA (Smith, 2001).
  - Una cosa fatta a metà rappresenta uno spreco di tempo che non ha portato feedback e tantomeno valore.
  - Lavorare troppo serve solo a creare più lavoro (errori), a essere meno produttivi (Schwartz, 2013), e a nascondere i problemi che altrimenti potrebbero venire risolti. Si veda la figura B.2 per notare come lavorare oltre un certo quantitativo non porti veramente benefici.
  - L'intelligenza emotiva (IE) è fondamentale per poter essere produttivi, mentre la dissonanza all'interno di un team rappresenta uno spreco poichè porta a una minore felicità e passione nel lavoro, con un risvolto drammatico per quanto riguarda la qualità del lavoro e l'apporto creativo delle persone facenti parte del gruppo (Goleman, 2016).

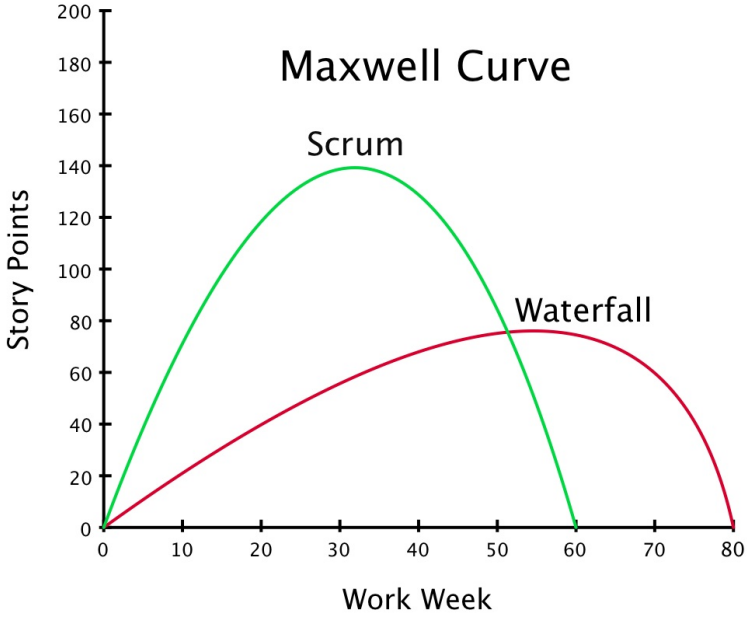


Fig. B.2: Curva di Maxwell
(Sutherland, 2012)

  - Rimandare la correzione di errori e la risoluzione dei problemi causa una maggiore mole di lavoro in futuro.

* Felicità, emozione che permette di prendere decisioni più sagge, di essere più creativi e anche più produttivi, perchè se si è felici si lavora più volentieri, oltre che meglio. Tenendosi alla larga dall'autocompiacimento e dalle bolle di felicità, essa rappresenta anche un indicatore predittivo (Figura B.3).

* Tempo (Appendice A), una risorsa inestimabile e limitata, non ci si può permettere di investire mesi di lavoro su qualcosa di cui non si conosce veramente il valore e che potrebbe rivelarsi inutile nel mondo reale. Bisogna cercare di arrivare velocemente al Minimum Valuable Product (MVP) per avere un confronto vicino nel tempo con i clienti.

* Priorità, spesso è il 20% delle caratteristiche di un prodotto a rappresentare l'80% del suo valore, principio conosciuto come la regola 80/20 o principio di Pareto (Lavinsky, 2014). Occorre identificarle e lavorare su di esse (Figura B.4).

* Storie: gli esseri umani tendono a pensare in termini relativi, pertanto descrivere una feature nella forma «*In veste di X, voglio Y, perciò farò Z*» aiuta a

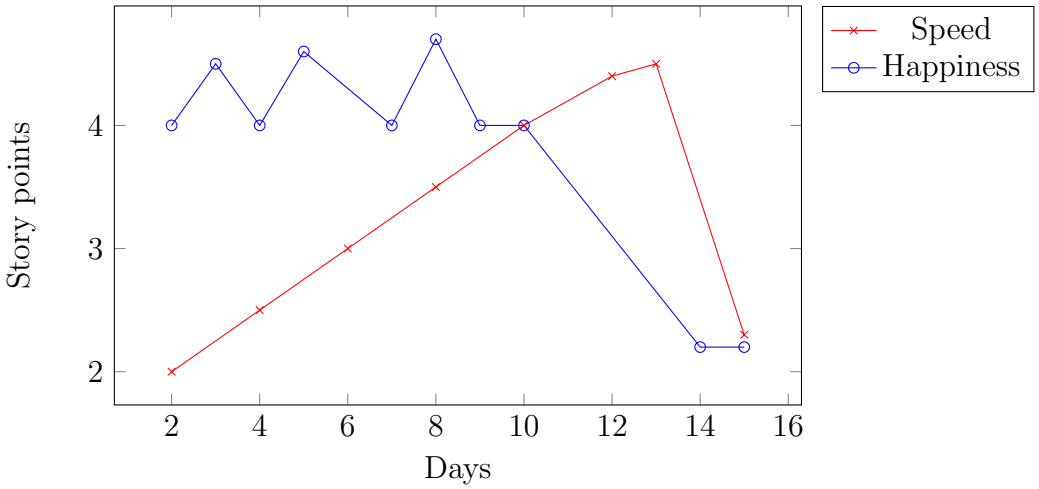


Fig. B.3: La felicità è un indicatore predittivo
(Sutherland, 2016)

comprendere il valore, la difficoltà (e di conseguenza il peso, o *Story points*) e la priorità. Ogni storia dovrebbe rispettare l'acronimo INVEST e quindi essere:

- Indipendente dalle altre storie.
- Negoziabile dal team.
- Valorizzante per il prodotto finale.
- Estimabile sia in termini di valore aggiunto che di peso.
- Snella, atomica.
- Testabile, il suo completamento deve essere insindacabilmente verificabile.

∗ Il gruppo (Team) di lavoro deve essere:

- Trascendente, bisogna avere un obiettivo che trascenda l'interesse del singolo.
- Autonomo, deve potersi gestire il lavoro internamente.
- Interfunzionale, ogni team deve avere al suo interno le risorse di cui necessita.
- Piccolo, per permettere la maggiore circolazione di informazioni possibile.
- Trasparente, chi ha qualcosa da nascondere intende fare i propri interessi.
- Non dovrebbe avere titoli, almeno all'interno.

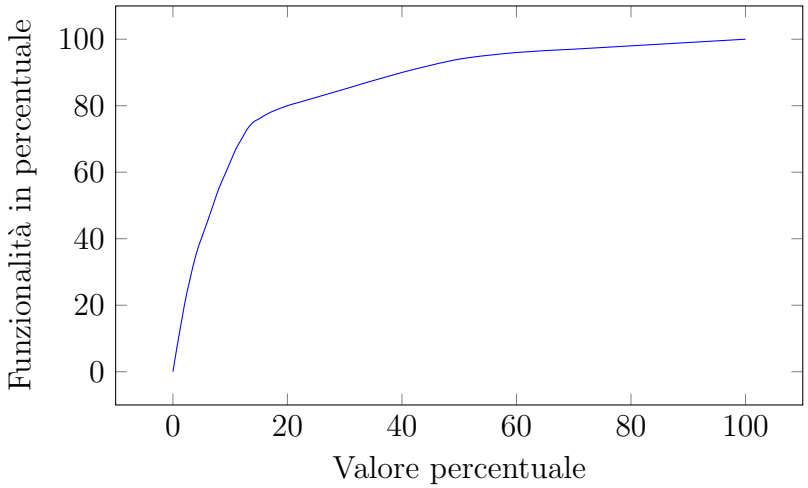


Fig. B.4: Curva del valore
(Sutherland, 2016)

B.2 Implementare Scrum

1. Nominare un *Product Owner*, ovvero colui che ha i contatti con il cliente, possiede l'intelligenza emotiva e appassiona i collaboratori, è disponibile al 100% per il team, tiene conto dei rischi e delle ricompense e possiede la visione del prodotto (Figura B.5)

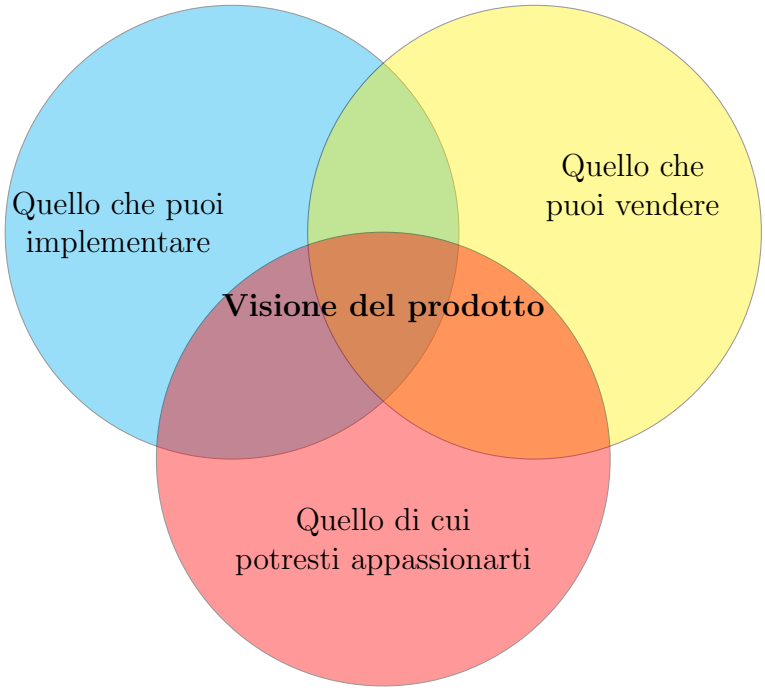


Fig. B.5: Visione del prodotto
(Sutherland, 2016)

2. Formare un Team con le caratteristiche descritte nei principi.

3. Nominare uno *Scrum Master*, una persona che conosca il metodo *Scrum* il cui compito è quello di focalizzare i team sull'obiettivo, riconoscere e rimuovere gli ostacoli alla produttività e che guidi i suoi collaboratori all'utilizzo di Scrum.

4. Creare e mettere in ordine di priorità un *Backlog*, ovvero un elenco di compiti con il loro peso (una misurazione relativa, non assoluta come per esempio potrebbero essere le ore, di quanto lavoro possa richiedere quel particolare compito), consultandosi con gli stakeholders.

5. Pianificare il primo *Sprint*, un intervallo di tempo fisso da una a quattro settimane, che inizia con l'operazione di selezionare quali compiti portare a termine per il prossimo rilascio incrementale e termina con la *Sprint Review* e la *Sprint Retrospective*. Conoscendo il totale degli *Story points* (la somma dei pesi dei compiti) e in quanto tempo sono stati guadagnati gli stessi, si può ricavare la velocità del team e stimare il tempo rimanente alla conclusione del progetto.

6. Eseguire ogni giorno una *Daily stand-up*, assemblea di quindici minuti che sostituisce ogni altra riunione di durata indeterminata, e dà modo ad ogni componente del gruppo di lavoro di rispondere a tre domande:

   - *≪Cos'ho fatto ieri per aiutare lo Sprint?≫*
   - *≪Cosa farò oggi per aiutare lo Sprint?≫*
   - *≪Cosa mi ha rallentato ieri? Che ostacoli ho incontrato?≫*

   Da queste sedute lo *Scrum Master* può comprendere meglio l'andamento del team e del progetto, individuando eventuali azioni correttive.

7. Rendere trasparente e visibile il lavoro svolto e che manca da svolgere. Per farlo si può usufruire di:

   ∗ Una *Scrum Board*, il principale manufatto del processo *Scrum* (Figura B.6). Nella prima colonna si trova l'intero *Backlog*, nella seconda i compiti da portare a termine nello Sprint corrente. Quando un componente inizia ad occuparsi di una Storia, la firma e la sposta nella terza colonna. Una volta finita viene posizionata nella quarta in attesa di affrontare la fase di test. Se passata, il compito viene messo nell'ultima colonna, altrimenti

torna alla precedente. L'utilità di questo oggetto deriva dall'immediatezza nel descrivere la situazione attuale del progetto e dell'occupazione dei membri del team, permettendo a management, Scrum Master e ai componenti stessi del gruppo di individuare percorsi critici, difficoltà, blocchi ed intervenire per risolvere le problematiche.

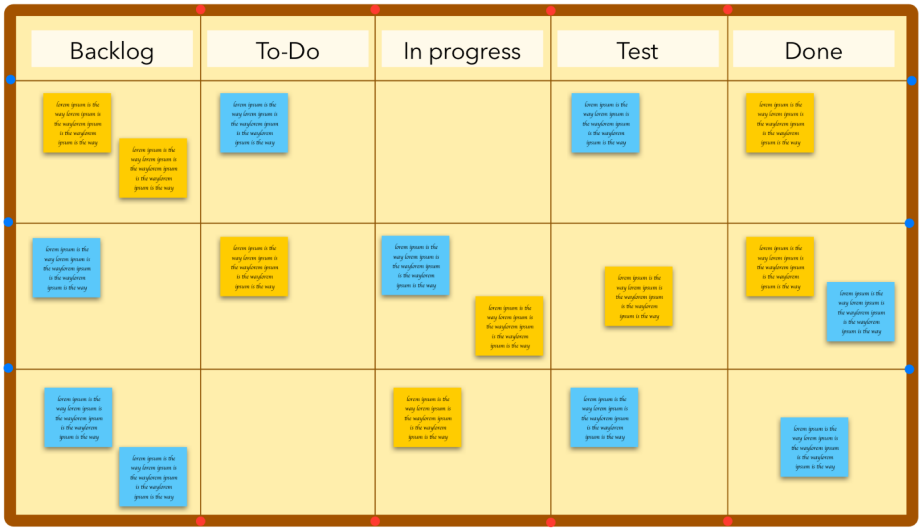


Fig. B.6: Scrum Board

∗ Un *Burndown Chart*, un grafico che descrive i punti rimanenti da raccogliere dal team per completare il prodotto. Idealmente dovrebbe essere una linea fortemente inclinata verso il basso, fino ad incontrare l'asse delle ascisse (Figura B.7).

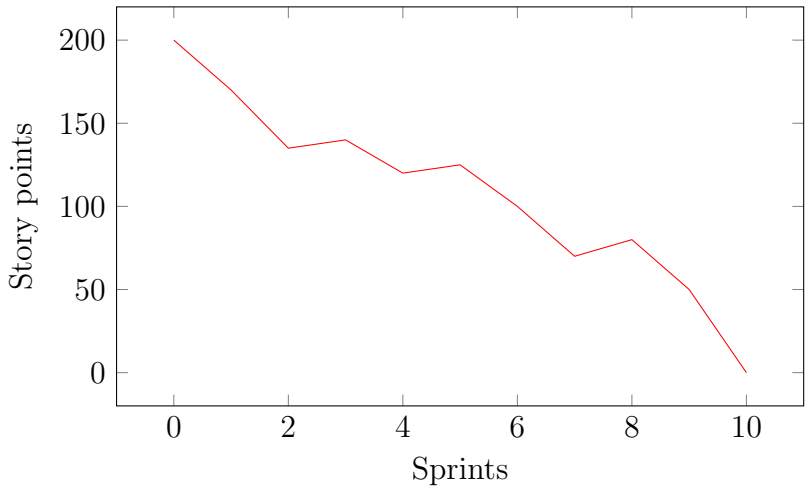


Fig. B.7: Burndown Chart

8. Effettuare una *Sprint Review* ogni fine Sprint, aperta a stakeholders, management e clienti, in cui viene mostrato ciò che è stato portato a termine nella sessione appena conclusasi, al fine di raccogliere un feedback sul lavoro svolto.

9. Effettuare una *Sprint Retrospective* dopo ogni Sprint Review, in modo di analizzare, internamente al team, lati positivi e negativi dello Sprint ed eventuali azioni correttive da adottare, in accordo al ciclo di Deming.

10. Ricominciare con un nuovo *Sprint Planning*.

Fonte principale: Fare il doppio in metà tempo. (Sutherland, 2016)

Appendice C

SOAP e REST

C.1 I Web Service

Un *web service* è un'applicativo on-line che permette l'interazione *machine-to-machine* al fine di ottenere o scambiare dei dati all'interno di Internet. Secondo la definizione del World Wide Web Consortium:

"Un Web Service è un sistema software progettato per supportare l'interoperabilità tra diversi elaboratori su di una medesima rete ovvero in un contesto distribuito" (Haas & Brown, 2004).

L'idea alla base dei *web service* è quella di poter offrire un'interfaccia software, accessibile attraverso la rete, che possa offrire risposte a delle interrogazioni più o meno semplici inviate da altre piattaforme. Tale interfaccia viene resa nota all'esterno grazie all'utilizzo di WSDL (Web Service Description Language) o alla pubblicazione di una documentazione delle API. Sono indipendenti dalla tecnologia e sono *loose coupling*, ovvero può non essere nota la struttura realizzativa interna.

C.2 SOAP

SOAP, acronimo di Simple Object Access Protocol, è il protocollo creato dal W3C al fine di gestire l'accesso ai *web service* da parte degli utenti o di altri computer. Il suo funzionamento non si basa sull'utilizzo di uno specifico protocollo per il trasporto di dati (**neutralità**, anche se comunemente viene usato HTTP), nè tantomeno sul linguaggio di programmazione utilizzato. L'unico cardine di SOAP è l'impiego di XML per descrivere la struttura dei servizi offerti e dei messaggi scambiati tra le applicazioni.

Le richieste SOAP sono tipicamente utilizzate per realizzare le *remote procedure call* (RPC), che sono caratterizzate da un'esecuzione sincrona, ovvero a ogni richiesta deve succedere una risposta.

C.2.1 Messaggio SOAP

Un messaggio SOAP è formato dai seguenti elementi:

- Un tag *envelope* che serve a racchiudere il documento e ad identificarlo come messaggio SOAP.
- Il blocco opzionale *header* in cui vengono inserite tutte le informazioni aggiuntive relative al servizio richiesto e all'interpretazione del messaggio.
- L'elemento *body* che contiene la richiesta/risposta e tutti i dati ad essa associati.
- Una sezione facoltativa *fault* in cui vengono scritte le informazioni relative ad eventuali errori in fase di elaborazione della risposta. Questo tag viene inserito dentro al *corpo* della risposta.

C.2.2 WSDL

Come detto precedentemente, WSDL è un'interfaccia per rendere noti all'esterno i servizi offerti dal *web service*. WSDL consiste in un documento XML in cui sono specificati, per ogni servizio:

- La posizione del servizio, ovvero l'URI attraverso cui può essere raggiunto.
- I metodi che il servizio offre e gli eventuali parametri di cui necessitano.

Per questo motivo WSDL presenta numerose analogie con l'interfaccia di una classe in linguaggio *object-oriented*, anche se il primo è indipendente dalla tecnologia utilizzata.

La struttura di un servizio all'interno di un documento WSDL è divisa in due parti:

1. Una parte detta *astratta* in cui sono definiti i metodi, i parametri e i tipi di dato utilizzati all'interno del servizio.
2. Una parte chiamata *concreta* nella quale sono definiti gli URI e i protocolli per accedere al *web service*.

C.3 REST

REST, acronimo di *REpresentional State Transfer*, è un insieme di requisiti per i *web service* che mira a sfruttare il protocollo HTTP e i suoi metodi per creare delle applicazioni CRUD (*Create*, *Read*, *Update*, *Delete*) in cui le risorse esposte non sono per forza codificati in un documento XML, ma possono essere definiti anche in altri formati, come per esempio JSON (Appendice D). Le caratteristiche di un *web service* di tipo REST sono le seguenti:

- L'utilizzo del protocollo HTTP e dei suoi metodi per distinguere le diverse azioni del servizio: POST per inserire, GET per leggere, PUT per modificare e DELETE per cancellare.
- L'impiego di rappresentazioni per incapsulare i dati (XML, JSON, ...).
- La mancanza di uno stato nell'interazione tra client e server (sistema *stateless*), caratteristica intrinseca di http.
- Si basa sul principio HATEOAS (*Hypermedia As The Engine Of Application State*), ovvero si utilizzano gli *hyperlink* per tenere traccia dell'avanzamento delle operazioni (per superare la limitazione di sistema *stateless*).

Tali caratteristiche hanno come vantaggio il fatto di conferire al *web service* una buona scalabilità nell'implementazione dei servizi ma, allo stesso tempo, rendono impossibile la creazione di un sistema *machine readable* in quanto non si dispone di uno schema fisso per la descrizione dei servizi. Infine, un web service che rispecchi tutte le caratteristiche sopraelencate viene definito RESTfull, in quanto non sempre ciò è possibile.

Appendice D

JavaScript Object Notation (JSON)

JSON è un linguaggio di interscambio dati basato sulla rappresentazione degli oggetti utilizzata nel linguaggio di programmazione JavaScript, anche noto come ECMAScript. È leggero, indipendente dal linguaggio di programmazione utilizzato dal sistema in cui viene impiegato, interpretabile dalla macchina ma comprensibile anche all'uomo.

Lo standard JSON (ECMA International, 2013) definisce poche e semplici regole per descrivere dati strutturati: un documento è infatti valido e ben formato se rappresenta un qualsiasi tipo di dato JSON e risulta conforme alla sintassi descritta nello standard.

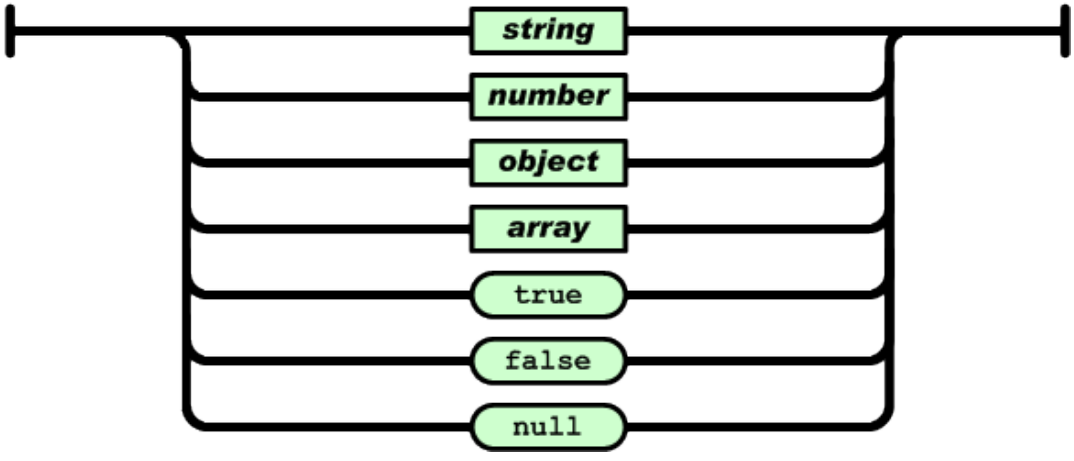


Fig. D.1: Struttura di un documento JSON ben formato

I tipi di dato definiti dallo standard sono quattro:

- *Numero*

- *Stringa*
- *Oggetto*
- *Array*

Inoltre sono presenti tre valori letterali che possono essere usati come tipo di dato: `true`, `false` and `null`.

Grazie alla possibilità di incapsulare oggetti e vettori, JSON può rappresentare stutture complesse come gli alberi, ma non è possibile creare dei riferimenti ciclici.

D.1 Numero

Ogni numero viene rappresentato in base 10, può essere preceduto dal segno meno [-], può rappresentare numeri decimali non periodici utilizzando il punto [.] come separatore e può essere caratterizzato da una potenza di 10 rappresentata da un numero intero (positivo o negativo) preceduto dalla lettera E oppure e. Un numero non può rappresentare valori come `Infinite` o `NaN`.

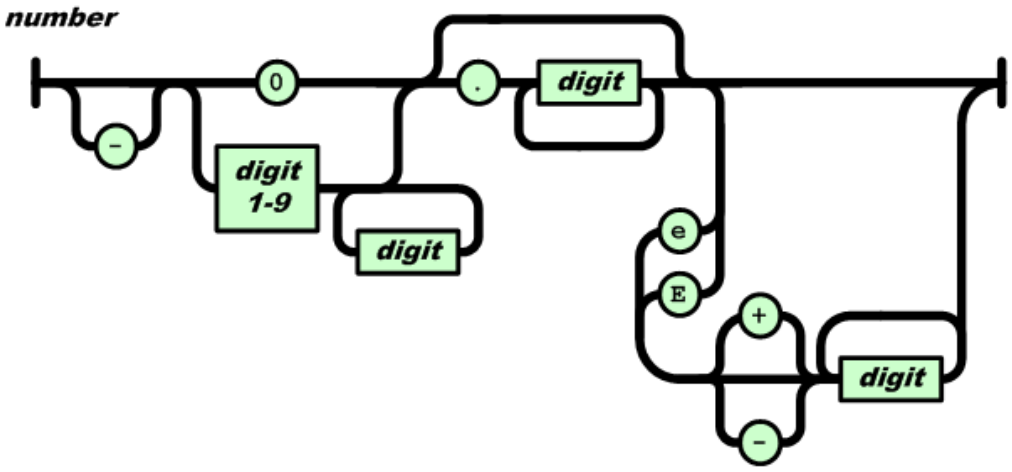


Fig. D.2: Struttura di un numero in JSON

D.2 Stringa

Le stringhe vengono rappresentate come una sequenza di caratteri *Unicode* delimitata da doppi apici ["] e ogni carattere può essere scritto all'interno degli apici, fatta eccezione per i doppi apici stessi ["], il *backslash* [\] e i caratteri speciali `U+0000` e

U+001F. JSON mette inoltre a disposizione alcune sequenze di escape per rappresentare determinati caratteri speciali:

\" rappresenta i doppi apici ["]

\\ carrisponde al carattere *backslash* [\]

\/ definisce lo *slash* [/]

\b rappresenta il *backspace* (U+0008)

\f indica il *form feed* (U+000C)

\n rappresenta il *line feed* (U+000A)

\r corrisponde al *carriage return* (U+000D)

\t rappresenta la tabulazione (U+0009)

Ogni carattere può essere rappresentato anche con il suo codice *Unicode* nella forma \uxxxx dove al posto di xxxx viene inserito il codice esadecimale a quattro cifre corrispondente.

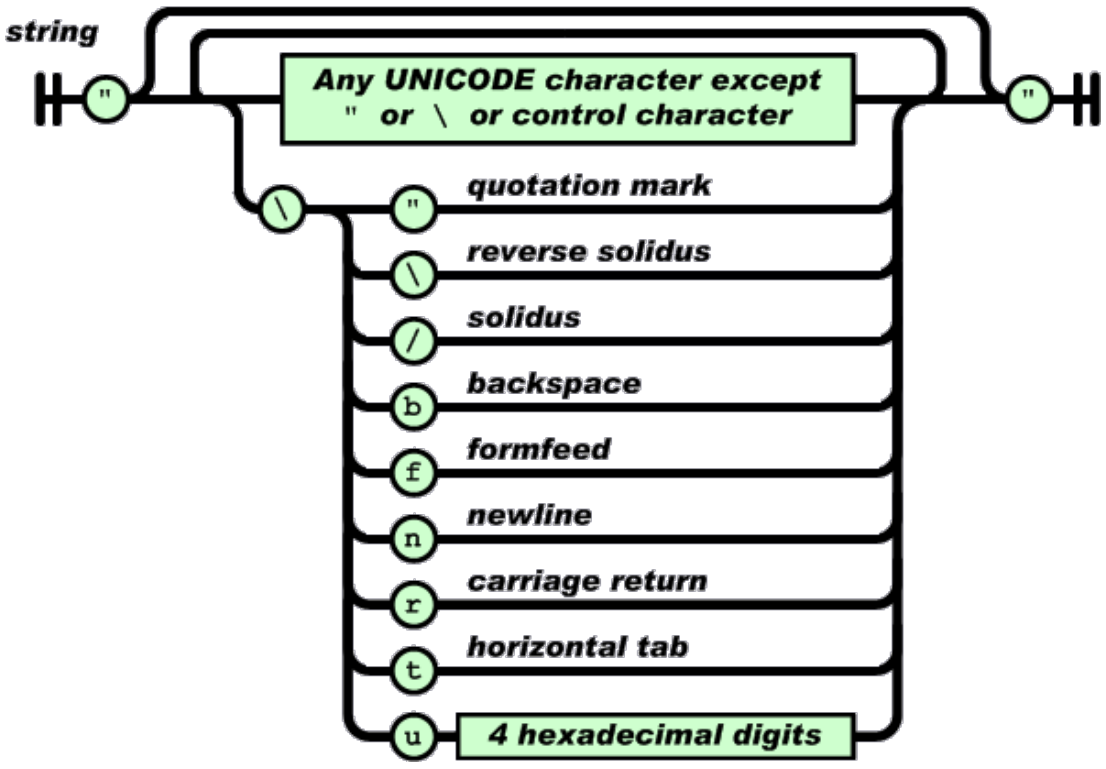


Fig. D.3: Struttura di una stringa in JSON

D.3 Oggetto

Un oggetto JSON è simile a ciò che nei linguaggi di programmazione viene chiamato *record*, *struct* o *map* e viene rappresentato da una coppia di parentesi graffe [{}], contenenti zero o più coppie nome - valore separate da una virgola [,]. Un nome è una stringa, mentre un valore è un qualsiasi tipo di dato (`true`, `false` e `null` inclusi); i due campi sono separati dai due punti [:].

Dato che ai valori si accede utilizzando il nome associato ad essi, l'ordine con cui essi vengono elencati non è influente e due oggetti con le coppie nome - valore permutate si possono considerare uguali.

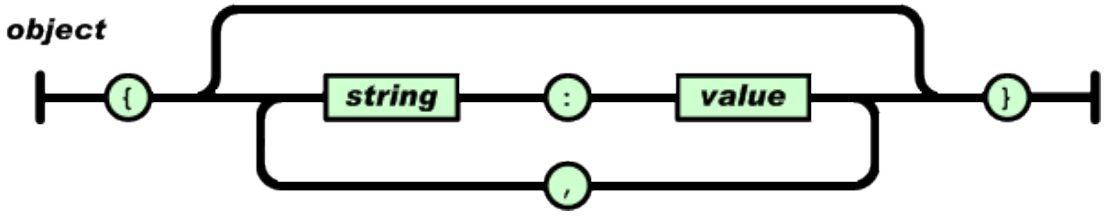


Fig. D.4: Struttura di un oggetto in JSON

D.4 Array

Un array JSON è la trasposizione dei vettori (o array o liste) dei linguaggi di programmazione e viene rappresentato da una coppia di parentesi quadre [`[]`] contenenti zero o più valori separati da una virgola [,]. Un valore, come in un oggetto, è un qualsiasi tipo di dato (`true`, `false` e `null` sempre inclusi).

Dato che ai valori si accede utilizzando la loro posizione, l'ordine con cui essi vengono scritti influisce sulla struttura dell'array.

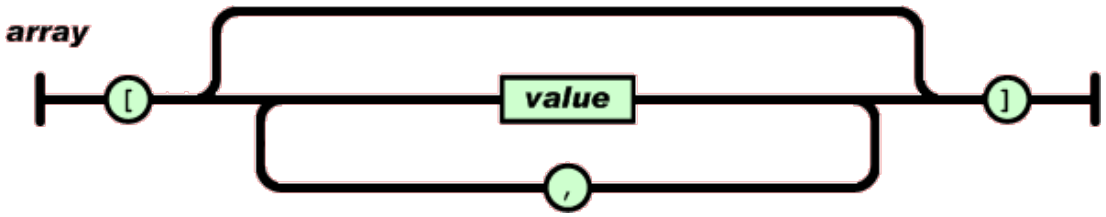


Fig. D.5: Struttura di un array in JSON

Appendice E

Database Distribuiti

E.1 Introduzione

Mentre con la notazione *Architetture distribuite* ci si riferisce a sistemi coinvolgenti un numero maggiore di due di processori autonomi, per *Database Distribuiti* si intende un'architettura in cui sono presenti almeno due server che possono anche svolgere transazioni autonome, ma che in alcuni casi devono interagire.

Appare necessaria la distinzione fra:

- DB Paralleli, ovvero basi di dati suddivise ma dipendenti.
- DB Replicati, ovvero più copie dello stesso insieme di dati separate.

Alla luce della precedente distinzione è possibile enunciare il cosiddetto Teorema CAP, detto anche Teorema di Brewer. Date le proprietà:

C. Consistency (Consistenza dei dati).

A. Availability (Disponibilità).

P. Partition Tollerance (Resistenza alle perdite)

Si ha che non è possibile ottenere la pienezza di queste contemporaneamente, ma solo di due (Figura E.1).

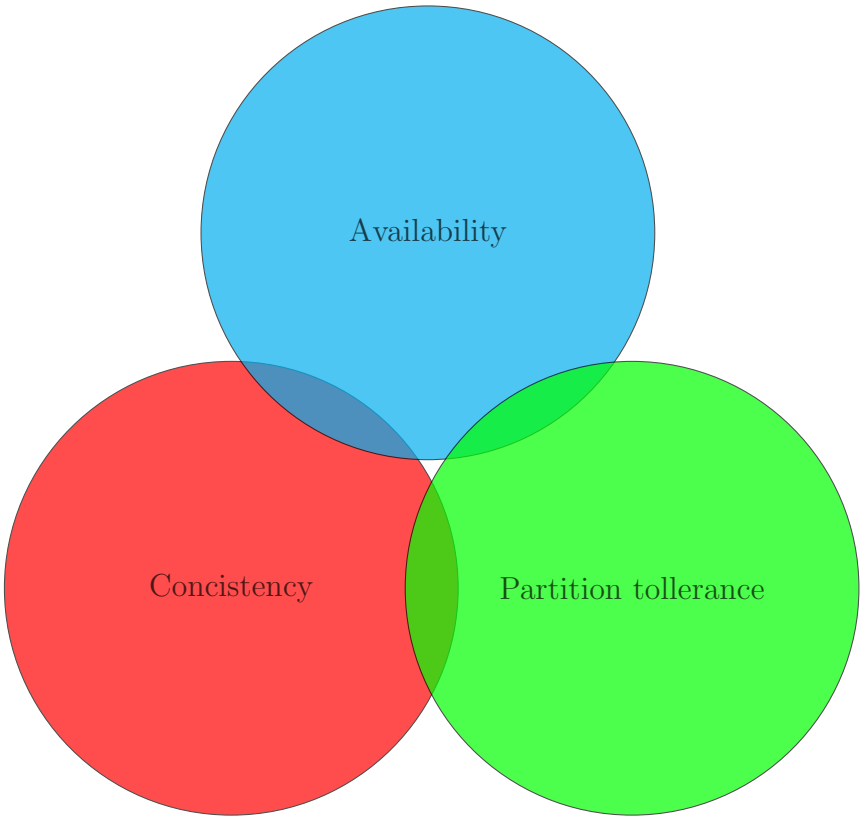


Fig. E.1: CAP Theorem

Pertanto spesso ci si rifà al principio BASE (BASIC AVAILABILITY SOFT STATE EVENTUALLY CONSISTENCY), per il quale si ottiene un approccio ibrido alle tre precedenti proprietà.

La distribuzione può essere più o meno trasparente, offrendo un servizio più o meno semplice da utilizzare. La trasparenza viene classificata secondo quattro livelli:

1. Trasparenza di frammentazione, se è come non ci fosse alcuna frammentazione.
2. Trasparenza di allocazione, quando è sufficiente conoscere in che modo i dati del database sono frammentati.
3. Trasparenza di linguaggio, se per accedere all'intero database distribuito è necessario conoscere i frammenti e la loro locazione.
4. Mancanza di trasparenza, nel caso in cui sia necessario conoscere la frammentazione, le locazioni e i linguaggi dei diversi DBMS.

È possibile, infatti, imbattersi in diversi DBMS in seguito all'esistenza di sistemi Legacy, ovvero sistemi magari anche datati che non è possibile o non è conveniente cambiare e aggiornare.

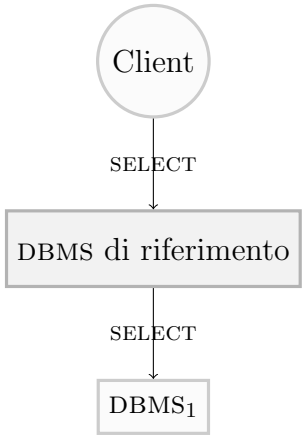


Fig. E.2: Richiesta remota

Una transazione su un database distribuito assume un significato più elaborato della semplice unità elementare di lavoro svolta da un'applicazione, tant'è che, nel contesto presentato, se ne possono individuare di diversi tipi. Prima di procedere con la classificazione, si desidera introdurre i concetti di DBMS di riferimento, ovvero quel DBMS a cui ci si rivolge per effettuare la transazione, e di DBMS locale, ovvero quel particolare nodo del sistema distribuito che ha in gestione un particolare sottoinsieme del database.

Il caso più semplice quando si opera su di un database distribuito si ha quando, effettuando una transazione composta da operazioni di selezione verso il proprio DBMS di riferimento, ci si riferisce indirettamente a un singolo DBMS locale. Si parla in questo caso di *Richiesta remota* (Figura E.2).

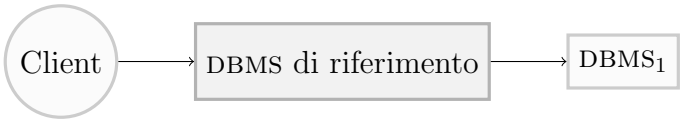


Fig. E.3: Transazione remota

Invece, nel caso in cui la transazione sia composta da diversi tipi di comandi ma comunque rivolti ad un unico DBMS, si parla di *Transazione remota*.

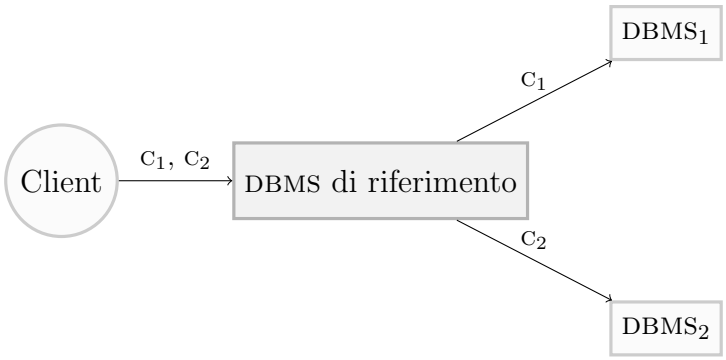


Fig. E.4: Transazione distribuita

Il terzo livello di complessità è rappresentato dalle *Transazioni distribuite* (Figura E.4), ovvero transazioni composte da diversi tipi di comandi (C_1, C_2) rivolti, ciascuno di essi, ad un singolo DBMS locale, e in toto a un numero maggiore o uguale a due.

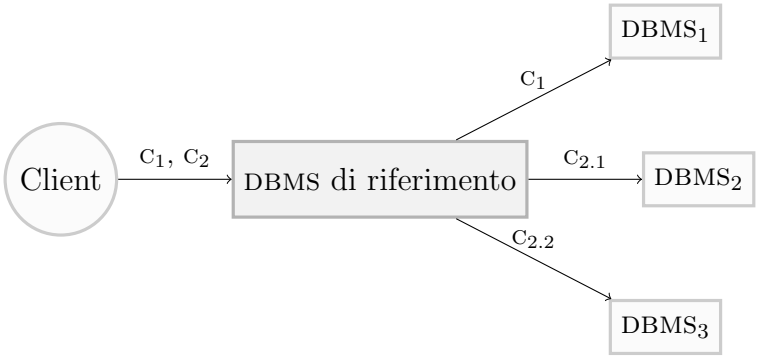


Fig. E.5: Richiesta distribuita

L'apice della complessità si ha con le *Richieste distribuite*, in cui ogni SOTTO-TRANSAZIONE può coinvolgere DBMS diversi (Figura E.5).

Appare chiara, dunque, la necessità di un *Gestore delle transazioni* (TRANSACTION MANAGER) poichè, per quanto ogni DBMS possa occuparsi della **consistenza** e della **durabilità** di ognuna di queste SOTTO-TRANSAZIONI, sono necessari dei protocolli diversi per garantire **l'atomicità** dell'intera transazione.

E.2 Commit a due fasi

Dopo aver decomposto e distribuito l'interrogazione svolta, il DBMS di riferimento definisce un ordine di esecuzione basato sul calcolo del COSTO di esecuzione di ogni SOTTO-TRANSAZIONE, dato dal costo computazionale e di trasmissione dati. Si rivela poi necessario il ruolo di SCHEDULER GLOBALE per serializzare e quindi evitare le anomalie classiche delle transazioni (Perdita di aggiornamento, Letture sporche, Letture inconsistenti, Aggiornamento fantasma). Infine, per evitare delle DEADLOCK distribuite (dentro uno stesso nodo o su nodi diversi), devono essere attuati dei meccanismi di timeout o di analisi delle sequenze di attesa (come sulle transazioni locali).

Come precedentemente introdotto, per garantire l'atomicità della transazione in toto, è necessario un protocollo diverso: il protocollo di **Commit a due fasi**. Il meccanismo di commit è composto da tre parti:

1. Fase di preparazione (Prepare phase), in cui il coordinatore globale domanda ai nodi partecipanti di confermare o annullare le relative transazioni locali; se almeno un nodo non può prepararsi, la transazione globale fallisce.

2. Fase di conferma (Commit phase), se tutti i nodi hanno portato a termine le proprie transazioni locali comunicandolo al DBMS di riferimento, quest'ultimo invita a confermare il commit.

3. Fase di conclusione (Forget phase), in cui il DBMS di riferimento può "dimenticarsi" della transazione, avendo concluso le operazioni per essa.

(Oracle, n.d.)

Si noti la peculiarità di questo meccanismo: non avviene un solo commit, bensì due; uno locale e uno globale di conferma. Si vuole evidenziare, inoltre, che questo protocollo non previene intrinsecamente dalla caduta di alcuni nodi o dalla perdita di messaggi, ma comprende la gestione di questi eventi. Si lascia però questa trattazione a testi più specifici.

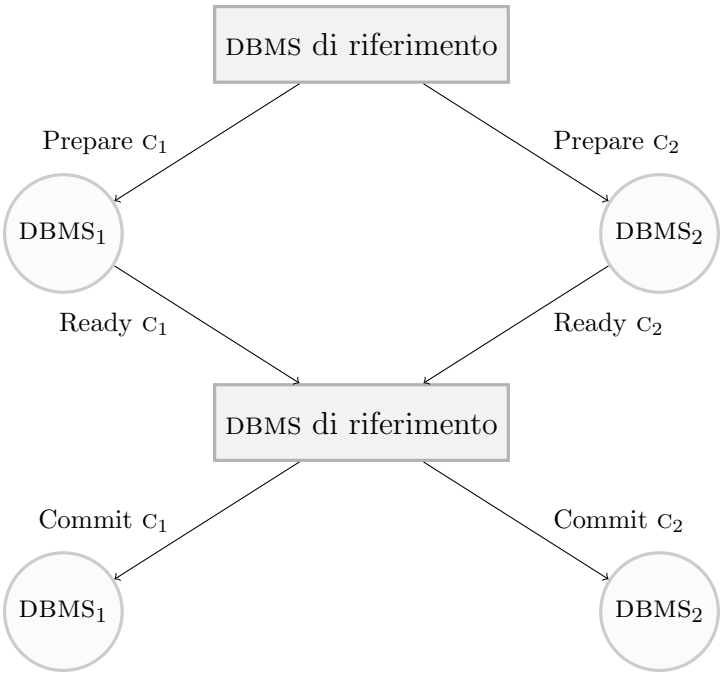


Fig. E.6: Commit a due fasi (Successo)

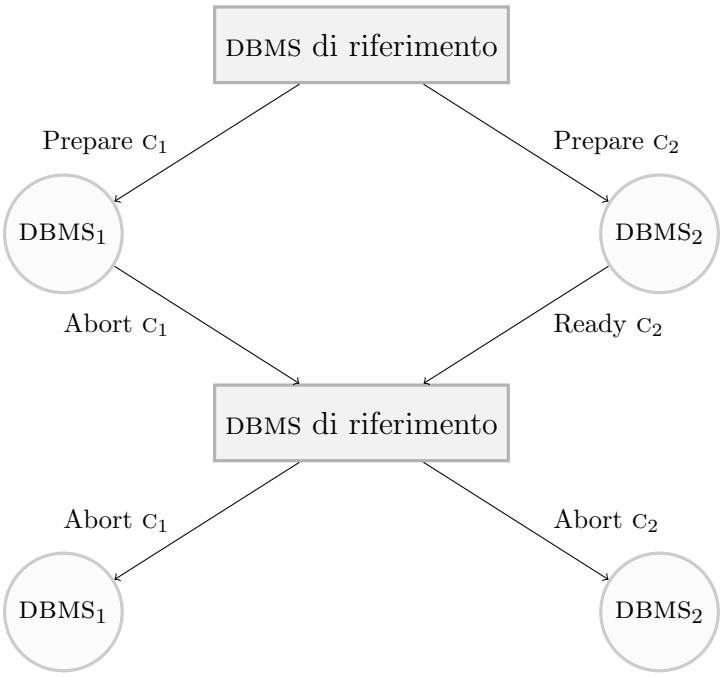


Fig. E.7: Commit a due fasi (Errore)

Appendice F

HTTPs

F.1 Introduzione

HTTPs è un protocollo di livello application che consente di trasmettere dei dati in modo sicuro attraverso una rete di computer e consiste nella comunicazione tramite il protocollo HTTP, utilizzando però una connessione criptata attraverso l'utilizzo di SSL/TLS. Solitamente HTTPs viene utilizzato per garantire la sicurezza dei dati all'interno di siti web governativi o di home banking, anche se, a partire dalla fine degli anni 2000, trova impiego in qualsiasi scenario in cui si vogliano proteggere i dati, come nel caso delle comunicazioni aziendali o nella protezione degli account utente.

Le funzionalità di base di HTTPs sono tre:

- Fornire l'autenticazione del sito web e del relativo server web per proteggere l'utente da possibili attacchi di tipo *man in the middle*, assicurando il fatto che si stia scambiando dati esattamente con il sito web voluto.
- Garantire la confidenzialità della comunicazione per proteggerla da eventuali operazioni di *eavesdropping*, ovvero un'intromissione nella comunicazione al fine di ascoltarla segretamente, certificando che la trasmissione dei dati non possa essere intercettata, o comunque risulti illeggibile ad un estraneo.
- Proteggere il contenuto dei messaggi da possibili azioni di *tampering*, attraverso le quali l'attaccante possa alterare la comunicazione, garantendo che i dati inviati vengano recapitati senza essere manomessi o, comunque, che eventuali modifiche possano essere rilevate.

F.2 Funzionamento

All'inizio di una trasmissione HTTPs (caratterizzata da un URI https del tutto identico a quello http, fatta eccezione per l'identificatore del protocollo, e dalla *default port* 443 TCP) il protocollo richiede al browser, o altro client HTTPs, di utilizzare un livello di cifratura TLS per proteggere la comunicazione, che verrà trasmessa come *application data* nello stream TLS. Per quanto riguarda le restanti funzioni HTTPs si comporta esattamente come HTTP.

TLS si occupa anche della fase di chiusura della trasmissione, permettendo sia al client che al server di considerare chiuso il canale sicuro, nel senso che da questo evento tutti i dati recapitati attraverso tale canale siano da considerarsi non sicuri e quindi non accettabili. Il client HTTPs che riceve una chiusura prematura (ovvero una chiusura dello *stream* TLS senza che lo *stream* HTTP sia terminato) deve segnalare un errore in quanto i dati ricevuti potrebbero non essere completi.

Per garantire l'autenticazione del sito web il client richiede una copia del certificato digitale del web server attraverso la quale viene controllata l'identità del Domain Name o, eventualmente, dell'IP utilizzato nell'URI.

F.3 HTTP

HTTP è il protocollo di livello application che viene comunemente usato per la navigazione nel web: la maggior parte dei siti internet, infatti, sono accessibili attraverso un server HTTP da cui vengono recuperati utilizzando un *client* HTTP, ovvero un comune *browser*. HTTP è uno dei protocolli più datati di Internet e le sue specifiche sono descritte nel RFC 1945.

Le principali caratteristiche di HTTP sono la sua leggerezza, che permette di inviare richieste e risposte mantenendo un header essenzialmente breve e con pochi campi obbligatori, e la genericità dello scopo dei messaggi che consente, tra le altre cose, di usare HTTP come protocollo di comunicazione tra applicazioni basate su altri protocolli (come SMTP, FTP o NNTP).

F.3.1 Richiesta

```
GET /folder/file.html HTTP/1.1
Host: agileclassroom.tk
User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/53.0
Accept: text/html, */*
Accept-Charset: iso-8859-1, utf-8;q=0.5, *;q=0.5
Connection: Keep-Alive
```

Fig. F.1: Esempio di richiesta HTTP

Come si può vedere nell'immagine (Figura F.1) una richiesta HTTP è formata da quattro parti:

1. La riga di richiesta in cui si trovano, in ordine: il metodo (GET, POST, HEAD, DELETE, ...), l'URI della risorsa a cui si vuole accedere e, infine, la versione del protocollo.

2. A partire dalla seconda riga vengono scritti i parametri dell'header della richiesta, di cui l'unico obbligatorio è *Host* per permettere l'utilizzo dei *virtual host*.

3. Dopo l'ultimo parametro, per separare l'header dal body, si trova una riga vuota formata dai caratteri ASCII "\n\r".

4. Dopo la linea bianca si trova il corpo della richiesta (*body*) che serve a contenere eventuali parametri utilizzati in metodi come POST.

F.3.2 Risposta

Quando un server riceve una richiesta HTTP la analizza e prova ad accedere alla risorsa specificata, restituendo in ogni caso una risposta come quella mostrata nell'immagine (Figura F.2).

```
HTTP/1.1 200 OK
Date: Mon, 3 May 2015 20:33:04 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 137
Last-Modified: Wed, 08 Jan 2010 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
  <head>
    <title>Pagina di esempio</title>
  </head>
  <body>
    Ciao a tutti, io sono una pagina di esempio :).
  </body>
</html>
```

Fig. F.2: Esempio di risposta HTTP

Tale risposta ha una struttura molto simile a quella della richiesta:

1. La prima riga è la riga di stato, formata dalla versione del protocollo, dal codice di stato HTTP di risposta, e dal mesaggio di stato.
2. Le sucessive righe compongono l'header della risposta, composto da campi come *Content-Length* che rappresenta la dimensione del body in Byte.
3. Per separare l'header dal body si usa sempre la sequenza "\n\r".
4. Dopo la riga vuota si trova il corpo della risposta che, come nel caso di richieste GET, rappresenta il contenuto della risorsa a cui si ha avuto accesso.

Il codice di stato è un numero a tre cifre che viene scelto nel seguente modo:

1xx) La risposta contiene un messaggio informativo.

2xx) Si ha avuto successo nell'esecuzione della richiesta, il più comune è **200 OK** che viene restituito quando tutto è andato a buon fine.

3xx) La richiesta è corretta ma il server non è in grado di rispondere; un esempio è il codice **302 Found** che fornisce l'URI a cui si trova la risorsa specificata (come nel caso dei *redirect*).

4xx) Nella richiesta è presente un errore del client, come nel caso del codice **404 Not Found** che indica che la risorsa non è stata trovata, oppure il codice **403 Forbidden** che viene restituito nel caso in cui non sia possibile accedere alla risorsa per motivi di permessi del file system, oppure ancora **418 I'm a teapot** inserito come Easter Egg nell'RFC 2324.

5xx) La richiesta non può essere soddisfatta a causa di un errore del server, come nel caso del codice **500 Internal Server Error** o **505 HTTP Version Not Supported**.

F.3.3 HTTP/2

Nonostante la versione di HTTP utilizzata tutt'ora sia la *1.1*, l'*Internet Engineering Task Force* ha proposto nel 2014 la nuova versione *2.0* che punta a migliorare alcuni aspetti del protocollo garantendo comunque la retro-compatibilità con HTTP/1.1:

- Sviluppare un meccanismo di negoziazione che permetta al client di scegliere il protocollo da utilizzare (anche diverso da HTTP).
- Migliorare la velocità di caricamento delle pagine web sfruttando tecniche come la compressione dei dati, la tecnologia *push*, il *pipeline* delle richieste e il *multiplexing* su una singola connessione TCP.
- Offrire diversi supporti per casi diversi casi d'uso (dimensioni della *viewport*, utilizzo di API o *web service*, utilizzo di *proxy* o *firewall*).

Le richieste HTTP/2 sono identiche a quelle HTTP/1.1 ma il nuovo protocollo permette al server di inviare risorse non ancora richieste (*push*) dal client in modo da fornire al web broser gli elementi necessari alla creazione della pagina web senza attendere il completamento dell'analisi della pagina.

Per migliorare ancora di più le prestazioni, HTTP/2 utilizza tecniche di framing dei messaggi al fine di ottenere un vero e proprio multiplexing delle richieste che, venendo distinte in *controllo* e *dati*, possono essere processate in parallelo.

F.4 SSL/TLS

SSL e il suo sucessore TLS sono dei protocolli di livello presentazione che vengono comunemente usati per trasferire in modo sicuro i dati attraverso reti TCP/IP, garantendo l'integrità, l'autenticazione e la cifratura dei pacchetti trasmessi. SSL non è più considerato sicuro e, perciò, viene usato solamente TLS.

Questi protocolli sono ampiamente utilizzati in servizi di email (principalmente PEC), messaggistica istantanea, VoIP e, soprattutto, nel protocollo HTTPS (HTTP over SSL).

F.4.1 Funzionamento

Una sessione TLS può essere divisa in tre parti:

- Negoziazione iniziale con scelta dei protocolli da usare.
- Scambio delle chiavi attraverso algoritmi come RSA, DH, PSK, ...
- Cifratura dei messaggi utilizzando tecniche a chiave simmetrica (per esempio RC4, DES, AES, ...) e verifica dell'autenticazione mediante quelle a chiave asimmetrica (RSA, DSA, ECDSA).

Riferimenti

Barale, L., Rascioni, S., & Ricci, G. (2014). *Futuro impresa 4*. Tramontana.

Baudelaire, C. (1859). *Il viaggio*. Revue française.

Berners-Lee, T. (1996, May). *Hypertext transfer protocol – http/1.0* (Request for Comments No. 1945). Retrieved 2017-06-03, from `https://tools.ietf.org/html/rfc1945`

Carroll, L. (1865). *Le avventure di alice nel paese delle meraviglie*.

Deming, W. E. (1950). Al management. Hakone Conference Center, Japan.

ECMA. (1999). *Ecmascript language specification*. ECMA Standardizing Information and Communication Systems. Retrieved 2017-06-01, from `https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf`

ECMA International. (2013). *The json data interchange format* (ECMA Standard No. 404). ECMA International Standardizing Information and Communication Systems. Retrieved 2017-06-01, from `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`

Einstein, A. (2000). *The expanded quotable einstein*. Alice Calaprice and Freeman Dyson.

Fielding, R. (1999, June). *Hypertext transfer protocol – http/1.1* (Request for Comments No. 2616). Retrieved 2017-06-03, from `https://tools.ietf.org/html/rfc2616`

Flask. (n.d.). Retrieved 2017-06-01, from `https://en.wikipedia.org/wiki/Flask_(web_framework)`

Goleman, D. (2016). *Essere leader, guidare gli altri grazie all'intelligenza emotiva*. BUR Rizzoli.

Greco, P. (2014, October 7). *Il tempo è un'illusione, per quanto tenace*. Retrieved 2017-05-29, from `http://www.unipd.it/ilbo/content/il-tempo-e-un%E2%80%99illusione-quanto-tenace`

Haas, H., & Brown, A. (2004, February 11). *Web services glossary* [W3C Working Group Note]. Retrieved 2017-05-31, from `https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice`

Khan, S. (2015, November 2). *Sal khan: Let's teach for mastery – not test scores* [Video file]. Retrieved 2017-05-28, from `https://www.ted.com/talks/sal_khan_let_s_teach_for_mastery_not_test_scores`

Lavinsky, D. (2014, January 20). Pareto principle: How to use it to dramatically grow your business. Retrieved 2017-05-30, from `https://www.forbes.com/`

sites/davelavinsky/2014/01/20/pareto-principle-how-to-use-it-to-dramatically-grow-your-business/#3de56e613901

Mark, G., Gudith, D., & Klocke, U. (n.d.). *The cost of interrupted work: More speed and stress* (Tech. Rep.). Department of Informatics, University of California, Irvine and Institute of Psychology, Humboldt University. Retrieved from https://www.ics.uci.edu/~gmark/chi08-mark.pdf

Oracle. (n.d.). *Oracle8i distributed database systems* (Documentation No. A76960-01). Retrieved 2017-05-30, from https://docs.oracle.com/cd/A87860_01/doc/server.817/a76960/ds_txns.htm

Ornella, S. (2008). *Pai fiòi dal kenedy.*

Percival, H. J. (2014). *Test-driven development with python.* O'REILLY.

Petrarca, F. (XIV secolo). *La vita fugge, et non s'arresta una hora - canzoniere.*

Proust, M. (1913 - 1927). *À la recherche du temps perdu.* Bernard Grasset.

Rescorla, E. (2000, May). *Http over tls* (Request for Comments No. 2818). Retrieved 2017-05-31, from https://tools.ietf.org/html/rfc2818

Retrofit, a type-safe http client for android and java. (2013). Retrieved 2017-06-05, from http://square.github.io/retrofit/

Ronacher, A. (2010-2017). Api [Computer software manual]. Retrieved 2017-06-01, from http://flask.pocoo.org/docs/0.12/api

Sant'Agostino d'Ippona. (400 ca.). *Confessiones.*

Schwartz, T. (2013, February 9). Relax! you'll be more productive. Retrieved 2017-05-30, from http://www.nytimes.com/2013/02/10/opinion/sunday/relax-youll-be-more-productive.html?pagewanted=all&_r=0

Smith, D. (2001, October). Multitasking undermines our efficiency, study suggests. , *32*(9). Retrieved 2017-05-30, from http://www.apa.org/monitor/oct01/multitask.aspx

Sutherland, J. (2012, February 14). The maxwell curve: Getting more production by working less! Retrieved 2017-05-30, from https://www.scruminc.com/maxwell-curve-getting-more-production

Sutherland, J. (2016). *Fare il doppio in metà tempo, puntare al successo con il metodo scrum.* Rizzoli Etas.

Vagrant. (n.d.). Retrieved 2017-06-01, from https://www.vagrantup.com