

UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Corso di Laurea Triennale in Matematica

Tesi di Laurea

LA MIA TESI
DI MATEMATICA

Relatore:

Prof. TAL DEI TALI

Laureanda:

GIOVANNA TESISTA

ANNO ACCADEMICO 2015-2016

Ai miei genitori
per non avermi tagliato i viveri

Indice

Elenco delle figure	vii
0.1 Introduzione	1
0.2 Primo approccio	2
0.2.1 Raccolta dei requisiti esplorativa	2
0.2.2 Studio di fattibilità	3
0.3 Pianificazione iniziale	6
0.4 Raccolta dei requisiti	8
0.4.1 Raccolta e analisi dei requisiti	8
0.5 Progettazione architetturale	14
0.5.1 Architettura del sistema	14
0.5.2 Metodologia di lavoro	17
0.5.3 Stima dei tempi, costi e risorse	17
0.6 Qualità del software	19
0.7 Database	25
0.7.1 Progettazione concettuale	25
0.7.2 Progettazione logica	26
0.7.3 Progettazione fisica e implementazione	29
0.7.4 Deploy locale	32
0.7.5 Testing	33
0.8 API web	35
0.8.1 Progettazione	35
0.8.2 Inizializzazione del contesto	39
0.8.3 Costruzione libreria di supporto	40
0.8.4 Implementazione API	45
0.8.5 Autenticazione	47
0.8.6 Testing	56
0.8.7 Deploy locale	57
0.9 Suddivisione dei compiti tra app e website	58
0.10 App	59
0.10.1 Design	59

0.10.2 Progettazione	65
0.10.3 Implementazione	67
0.10.4 Testing	69
0.11 Website	72
0.11.1 Struttura	72
0.11.2 Design	72
0.11.3 Implementazione	75
0.11.4 Testing	85
0.11.5 Deploy locale	86
0.12 Deploy	89
0.12.1 Deploy del database	89
0.12.2 Deploy delle API e website	90
0.12.3 Pubblicazione dell'app	90
0.13 Conclusioni e sviluppi futuri	92
Bibliografia	95

Elenco delle figure

1	Appunti della raccolta requisiti esplorativa	3
2	Modello waterfall (a sinistra) vs iterativo (a destra)	6
3	Introduzione del prototyping	7
4	Pianificazione iniziale	7
5	Parte del documento dei requisiti	9
6	Frammento del data dictionary del progetto	10
7	Processo di risk management	12
8	Modello a spirale	13
9	Diagramma di contesto del sistema	15
10	Struttura dell'architettura REST	16
11	Esempio di file json	16
12	Caratteristiche della qualità	22
13	Verifica e validazione	24
14	Schema ER	26
15	Esempio di tabella associativa usata per modellare una relazione N a N	27
16	Frammento della tabella delle frequenze	27
17	Frammento della tabella dei volumi	28
18	Esempio di tabella degli accessi per l'analisi delle ridondanze	28
19	Schema ER ristrutturato	29
20	Utilizzo di Docker Desktop con i container usati nel progetto	33
21	Attività di testing sul database con pgAdmin4	34
22	Esempio di sequenza di commit, ciascuno con il proprio identificativo	36
23	Esempio dell'uso di git in un progetto condiviso	37
24	Operazioni principali durante l'utilizzo di git	37
25	Creazione di un nuovo access token (successo)	49
26	Esecuzione di una richiesta protetta (successo)	52
27	Logout ed eliminazione token (successo)	53
28	Email di recupero password	55
29	Console di debug per Flask	56

30	Utilizzo del client REST Insomnia	56
31	Grafo dei concetti	61
32	Esempio di sidebar	62
33	Wireframe dell'app	63
34	User-centered design	64
35	Gerarchia schermata "Home_no_attivita"	65
36	Dati utilizzati dai componenti di "Home_no_attivita"	66
37	Struttura navigazione principale	66
38	Prima versione della schermata - i dati sono finti	67
39	I dati sono caricati dall'API	68
40	Scelta colori e dettagli grafici	69
41	Testing - Dispositivo di sviluppo	70
42	Testing - dispositivo mobile	70
43	Esempio di navigazione con sidebar	73
44	Esempio di mockup - pagina persone	74
45	Esempio di mockup - struttura pagina persone	75
46	Testing con il browser <i>Google chrome</i> durante lo sviluppo del website	86
47	Persona	86
48	Login	87
49	Recupero password	87
50	Nuova persona	88

0.1 Introduzione

La Protezione Civile del comune di San Vito al Tagliamento [?] attualmente utilizza esclusivamente moduli cartacei per la gestione delle proprie attività, il tracciamento della partecipazione delle squadre e il mantenimento delle informazioni riguardanti il proprio inventario.

Lo scopo di questo progetto è la realizzazione di un sistema informatico che possa contribuire a migliorare l'efficienza, la sicurezza e la semplicità di gestione della parte burocratica. Inoltre, un'integrazione automatica dei dati e delle richieste migliora le capacità organizzative, in quanto semplifica la comunicazione tra i membri, rimuove l'onere dei capisquadra di ricordare precisamente tutte le informazioni necessarie (come l'elenco delle presenze e dei materiali) e fornisce una dashboard unica in cui è possibile visualizzare tutte le informazioni di cui si può aver bisogno.

In questa tesi verrà mostrato l'intero processo di sviluppo, a partire dal primo approccio con il committente e finendo con la pubblicazione del sistema. Verrà data particolare attenzione ai processi secondari e di gestione, i quali non producono software, ma sono necessari all'organizzazione interna del lavoro. Verranno anche presentati numerosi approfondimenti su processi e tecnologie specifiche applicati durante lo svolgimento del progetto.

0.2 Primo approccio

Il primo approccio consiste nella conoscenza reciproca tra committente e gruppo operativo.

Questa è una delle fasi più critiche perché da essa si determina la volontà di iniziare il progetto: si stabiliscono, a grandi linee, i requisiti che la soluzione dovrà avere, per capire quale sia la strada migliore affinché il committente possa risolvere il suo problema. Infatti, mettendo al primo posto la soddisfazione del cliente, non è scontato che la costruzione di un sistema informatico sia la soluzione migliore, né è scontato che il gruppo di lavoro disponga delle conoscenze e risorse adeguate allo sviluppo di tale sistema.

0.2.1 Raccolta dei requisiti esplorativa

Lo scopo della raccolta dei requisiti esplorativa è presentare al gruppo di lavoro il dominio applicativo¹ e il problema che si intende affrontare. Non è importante scendere nei dettagli: quello che è importante in questa fase è capire quale sia la soluzione ottimale per raggiungere la soddisfazione del cliente.

In genere viene effettuata mediante un singolo incontro preliminare, possibilmente di persona, tra un rappresentante del committente e il responsabile del gruppo operativo per quanto riguarda le relazioni esterne. Essendo molto sommaria, non necessita di una presenza massiccia, lasciando liberi gli altri membri del gruppo.

La discussione è informale e priva di schemi. Inizialmente il committente viene lasciato libero di presentare il contesto e le difficoltà attuali. Proseguendo nella conversazione, si chiede in linea generica quali siano le soluzioni precedenti che il cliente ha provato (se ce ne sono), quale sia la soluzione che il committente si immagina (se ce n'è una) e quali siano le sue aspettative su di essa. A questo punto il rappresentante del gruppo di lavoro dispone di abbastanza informazioni per poter formulare alcune soluzioni ipotetiche, e le presenta al committente, discutendone pregi e difetti. È importante concordare sull'ordine di grandezza di tempi, risorse e costi riguardanti il progetto, e se possibile elencare tutti i requisiti più stringenti (ad esempio la consegna rigorosamente entro una certa data, se richiesta).

¹Il *dominio applicativo* è il contesto in cui il sistema opera.

Il tutto è effettuato in via approssimativa, quindi non è necessario avere conoscenze tecniche troppo specifiche. È però fondamentale che il rappresentante del gruppo operativo abbia conoscenze adeguate in modo da poter formulare soluzioni ragionevoli.

Uno dei modi migliori di documentare l'incontro è scrivendo su carta in formato di appunti. È importante che siano sufficientemente comprensibili e coprano tutti gli argomenti affrontati. Un'altra tecnica è la registrazione dell'incontro, però rischia di mettere il committente a disagio, impedendogli di essere naturale e mettendogli timore di dire cose stupide.

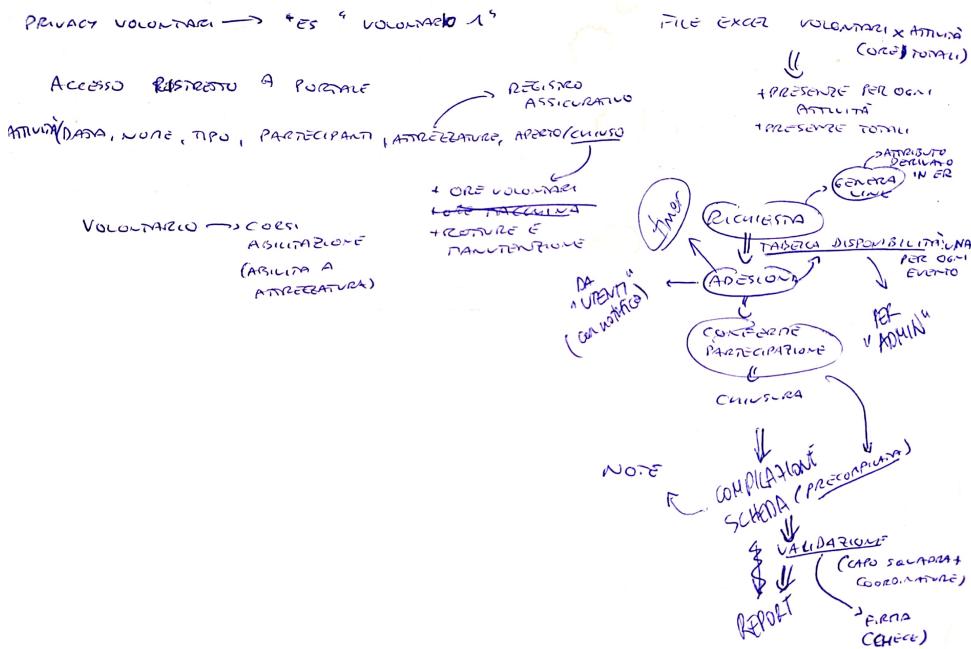


Figura 1: Appunti della raccolta requisiti esplorativa

0.2.2 Studio di fattibilità

Una volta completato l'incontro, il responsabile per le relazioni esterne si riunisce con l'intero gruppo di lavoro e presenta a tutti il progetto.

Il contributo tecnico altamente specifico di tutti i membri consente di effettuare, in primo luogo, lo studio di fattibilità ed una prima stima di tempi, costi

e risorse. Infatti, in particolare nel caso di progetti svolti in gruppi ristretti, le richieste temporali e tecniche del progetto devono collimare con le disponibilità di tutti i membri del gruppo, altrimenti si rischia di ottenere un progetto incompleto.

La modalità migliore di discussione è, ancora una volta, quella di effettuare un incontro "live". Una documentazione scritta richiederebbe un dispendio di tempo ed energie molto più elevato e non dà alcun vantaggio concreto.

Nel caso di questo progetto lo studio di fattibilità si è svolto in videoconferenza e ha prodotto i seguenti risultati:

- Il progetto è completamente fattibile con tempi, risorse e costi adeguati.
- Il tempo di sviluppo indicativo è: 6 mesi.
- Non sono richieste risorse aggiuntive oltre quelle già presenti nel gruppo di lavoro.
- Non sono previsti particolari oneri finanziari a carico del committente o del gruppo di lavoro.

Una volta svolto lo studio di fattibilità il committente viene contattato per informarlo dell'esito e, nel caso sia positivo, si conferma la volontà reciproca di proseguire nel progetto.

Approfondimento: Come si valuta il successo?

Ci sono molti modi di definire il *successo*. Ciò dipende sia dagli obiettivi personali, sia dal contesto in cui si parla. Alcuni esempi di caratteristiche del successo potrebbero essere: il profitto, la soddisfazione personale, il contributo ad una causa etica...

Una definizione più formale di *successo*, per quanto riguarda la creazione di un sistema informatico, è la seguente:

Svolgere il lavoro

- *in time* e *on schedule*: concludendo entro il tempo limite e rispettando la pianificazione;
- *on budget*: rispettando i costi prestabiliti;

- in accordo con *requirement* ed *expectation*: deve fare quello che si era stabilito inizialmente, ma anche soddisfare le aspettative del committente;
- con un team felice: perché è importante che anche i membri del team siano contenti e soddisfatti del lavoro svolto!

Possiamo quindi vedere il successo non come un concetto unico, ma come la realizzazione di più obiettivi, che a volte possono persino essere contrastanti. Una delle maggiori difficoltà nel portare avanti un progetto è quella di trovare il giusto tradeoff per affrontare al meglio tutte le situazioni impreviste, in modo da trovare la conclusione migliore possibile.

0.3 Pianificazione iniziale

Per iniziare a lavorare sul progetto, è fondamentale decidere fin da subito quale sarà il modello di sviluppo adottato, in quanto esso ci dà gli strumenti per dividere il lavoro in task, organizzarli gerarchicamente e metterli in relazione tra loro [? ?].

Le principali tipologie di modello sono: *waterfall* e *iterativo*. Nel modello *waterfall* il processo di sviluppo si divide in fasi (requirements, specification, design, implementation, testing...) che vengono svolte in un ordine ben preciso, e gli output di una fase costituiscono l'input della successiva. Per dare inizio ad una fase bisogna attendere il completamento della precedente. Nei modelli *iterativi* le fasi sono molto brevi e vengono ripetute iterativamente, in modo da aggiungere un po' alla volta nuove caratteristiche e funzionalità. Il modello *waterfall* fornisce migliori garanzie per quanto riguarda la prevedibilità, ma necessita di conoscere fin dall'inizio in modo preciso tutti i requisiti e pone molte difficoltà nel caso di cambiamenti in corso d'opera.

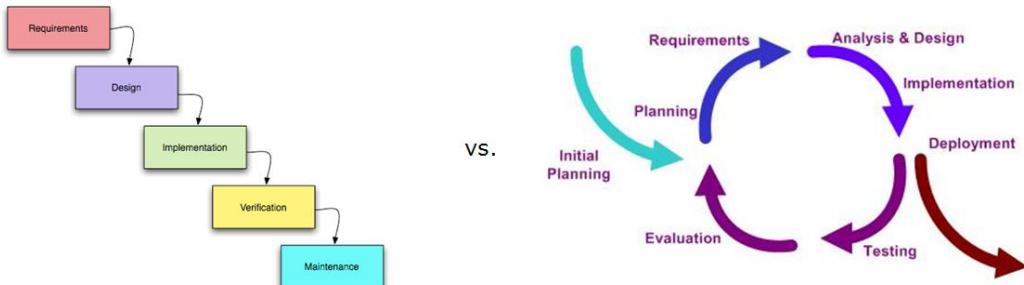


Figura 2: Modello waterfall (a sinistra) vs iterativo (a destra)

Una tecnica di supporto è il *prototyping*: la costruzione di prototipi dimostrativi a basso costo che permettono la prova delle nuove funzionalità prima della loro effettiva implementazione. Il prototyping è molto utile quando si parte da specifiche incomplete (*evolutionary prototyping*) oppure quando si vuole visualizzare un concetto nuovo (*throw-away prototyping*).

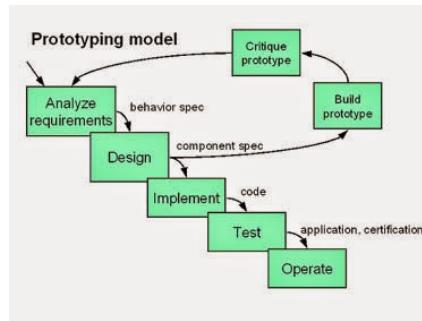


Figura 3: Introduzione del prototyping

Il progetto nel generale utilizza una metodologia mista: c'è una sequenza di operazioni ben definita stile waterfall, in cui però le singole operazioni vengono ottimizzate con tecniche iterative e di prototyping.

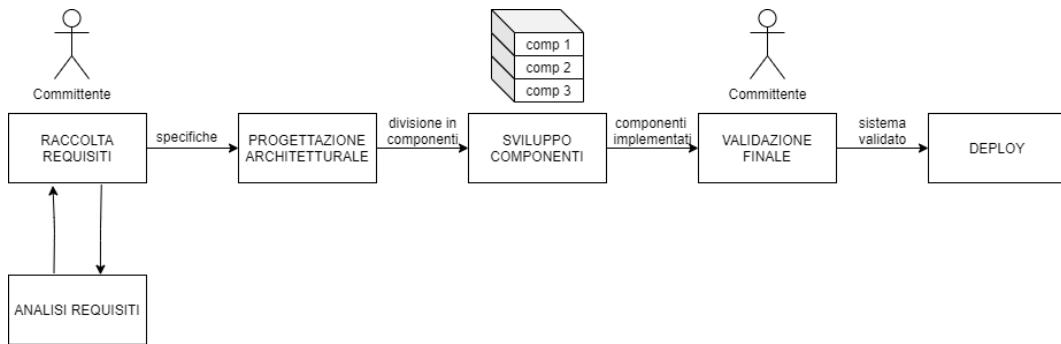


Figura 4: Pianificazione iniziale

0.4 Raccolta dei requisiti

La raccolta dei requisiti [?] si prefigge di elencare *precisamente* tutte le caratteristiche *funzionali* e *non funzionali*² che il sistema dovrà avere. Questa fase serve per avere un'idea il più chiara possibile di quello che il committente vuole, in modo da *minimizzare il rischio*³ di produrre un sistema che non incontri le sue aspettative.

0.4.1 Raccolta e analisi dei requisiti

La raccolta dei requisiti per il progetto è stata effettuata mediante una serie di incontri tra tutti i membri del gruppo di lavoro (in simultanea) e un membro responsabile/futuro utilizzatore del sistema per conto del committente.

Gli incontri, con il passare del tempo, sono diventati sempre più strutturati: mentre all'inizio era il committente a spiegare liberamente quelle che sono le sue richieste, poi erano i singoli membri del gruppo di lavoro a porre domande specifiche, mirate ad approfondire gli aspetti più critici per lo sviluppo del sistema.

Dopo il termine di ogni incontro sono stati organizzati dei meeting tra tutti i membri del team in modo da discutere, analizzare e sintetizzare quanto appreso, e di conseguenza preparare le domande per l'incontro successivo con il committente.

²Le caratteristiche funzionali legano gli input del sistema ai suoi output (ciò che il sistema fa). Le caratteristiche non funzionali sono molto più generiche e possono includere sia caratteristiche del sistema (ad esempio: tempi di risposta, tolleranza agli errori), sia vincoli esterni (ad esempio: legislativi, amministrativi, finanziari).

³Si riferisce all'attività di *risk management*

1.2 Dominio applicativo

Come anticipato, si vuole modellare un sistema unificato per la pianificazione delle attività di competenza dell’ente Protezione Civile di San Vito al Tagliamento e per la raccolta delle relative disponibilità del personale. Segue una schematica descrizione del funzionamento di tale sistema.

- Ogni attività è identificata univocamente da un codice interno ed è caratterizzata dalle seguenti informazioni: tipo (scelto da un elenco modificabile), comune dove si svolge, descrizione, causa, data e ora di inizio e di fine, partecipanti, responsabile (unico), attrezzatura necessaria, eventuali note ed attivazione della SOR (con eventuale ammontare di ore per cui questa è stata attiva).
- Possono esistere attività per le quali si vuole tener traccia del limite di età minima e/o massima. Un utente ha la possibilità di dare la propria disponibilità anche se non rispetta i requisiti di età.

Figura 5: Parte del documento dei requisiti

Approfondimento: Requisiti di qualità

Alla scrittura di requisiti di qualità viene spesso data poca importanza, in quanto vista come perdita di tempo poco utile ai fini dell’implementazione. Al contrario, il documento dei requisiti è il fulcro di tutta l’attività di sviluppo; infatti viene usato in ogni fase per capire cosa il committente *vuole esattamente* dal sistema in produzione.

Il documento dei requisiti [?] è l’input primario per la fase di definizione delle specifiche. Viene poi usato in fase di progettazione per sviluppare il sistema che meglio si adatta al committente e al contesto. È fondamentale per identificare le *caratteristiche della qualità* importanti nel contesto di utilizzo. È utilizzato in fase di verifica per assicurarsi che il sistema faccia tutto ciò che deve (o non deve). Infine, può risultare tra i documenti legalmente vincolanti più influenti su cui si basa il contratto tra committente e sviluppatore.

Risulta quindi necessario avere un criterio oggettivo e sistematico per la creazione di requisiti di qualità. Un buon documento dei requisiti ha le seguenti caratteristiche:

- *correctness*: fornisce le funzionalità che supportano al meglio il committente;
- *consistency*: non ci sono conflitti tra requisiti;
- *completeness*: sono incluse tutte le funzionalità richieste;
- *comprehensibility*: è facilmente comprensibile da tutti gli stakeholders;

- *realism*: i requisiti sono fattibili;
- *traceability*: si comprendono bene sia i collegamenti tra richieste e requisiti, sia i link inter-requisito;
- *adaptability*: può essere modificato facilmente per venire incontro a nuove esigenze;
- *verifiability*: i requisiti possono essere verificati in modo oggettivo.

Il formato principale per il documento dei requisiti è testuale. Per questo è importante evitare ambiguità (la stessa parola ha significati diversi) e eccessiva flessibilità (lo stesso significato viene espresso con parole diverse). Un valido strumento in questi casi è il *data dictionary*: un breve dizionario che spiega tutti i termini tecnici utilizzati e eventuali termini che possono risultare ambigui.

- ATTIVITÀ – Un qualsiasi evento organizzato dalla Protezione Civile. Questo concetto è soggetto ad evoluzione nel tempo: passa dallo stato di *pianificata*, durante il quale vengono prima raccolte le adesioni e poi viene messa in pratica, allo stato di *completata*, al momento della validazione da parte del responsabile.
- PERSONA – Membro dell'ente Protezione Civile. Si specializza nei tipi (disgiunti) *volontario*, *coordinatore* e *caposquadra*.
- PERSONA ESTERNA – Persona non membro della Protezione Civile, ma che collabora allo svolgimento di almeno una attività.
- PARTECIPANTE – Colui che partecipa ad un'attività; l'insieme dei partecipanti è dato dall'unione di PERSONE e PERSONE ESTERNE.

Figura 6: Frammento del data dictionary del progetto

Per superare la linearità del linguaggio naturale è spesso utile utilizzare forme grafiche come: tavole, flowchart, schemi, disegni, pseudolinguaggi...

Uno dei meccanismi per la verifica dei requisiti è la generazione di *scenari*: si creano dei personaggi (stereotipi dei futuri utilizzatori), un contesto d'uso e una problematica che si vuole risolvere/azione da eseguire. Si fanno quindi interagire gli utenti con il sistema, in modo da mostrare al committente un esempio pratico di come il sistema si comporterà e verificare che faccia ciò che il committente si aspetta. Con l'utilizzo di *prototipi* reali o in realtà virtuale, è possibile attuare gli scenari in modo da dare al committente un'esperienza ancora più realistica ed immersiva.

Approfondimento: Risk management

L'attività di *risk management* [?] si occupa di identificare i rischi e ideare piani per evitarli.

Un *rischio* è una probabilità dell'avvenimento di un evento negativo. Esso quindi è naturalmente caratterizzato da una *probability* (very low < low < medium < high < very high) e da una *severity* (insignificant < tolerable < serious < catastrophic). I rischi riguardano tutti gli aspetti del progetto: sia quelli strettamente legati all'implementazione, sia quelli interni al gruppo di lavoro, sia quelli esterni (legati al committente o indipendenti da esso).

Le fasi del risk management sono:

1. *risk identification*: fornisce una lista dei rischi (*risk list*);
2. *risk analysis*: assegna una priorità ai rischi (*prioritized risk list*);
3. *risk planning*: pianifica le azioni di contrasto ai rischi (*risk avoidance* e *contingency plan*);
4. *risk monitoring*: monitora i rischi già identificati e l'insorgenza di nuovi (*risk assessment*).

Le azioni intraprese dal risk management sono:

1. *avoidance*: riduce la probability del rischio;
2. *minimization*: riduce la severity del rischio;
3. *contingency*: attua contromisure al verificarsi del rischio.



Figura 7: Processo di risk management

È importante valutare tradeoff ottimali in modo che le azioni intraprese contro i rischi non abbiano controindicazioni peggiori dei rischi stessi (ad esempio, per minimizzare il rischio di produrre codice di bassa qualità si finisce per dilazionare troppo i tempi di implementazione e conseguentemente concludere in ritardo).

Il *metodo a spirale* [?] è un particolare metodo iterativo che basa le azioni da intraprendere sulla continua valutazione e minimizzazione dei rischi.

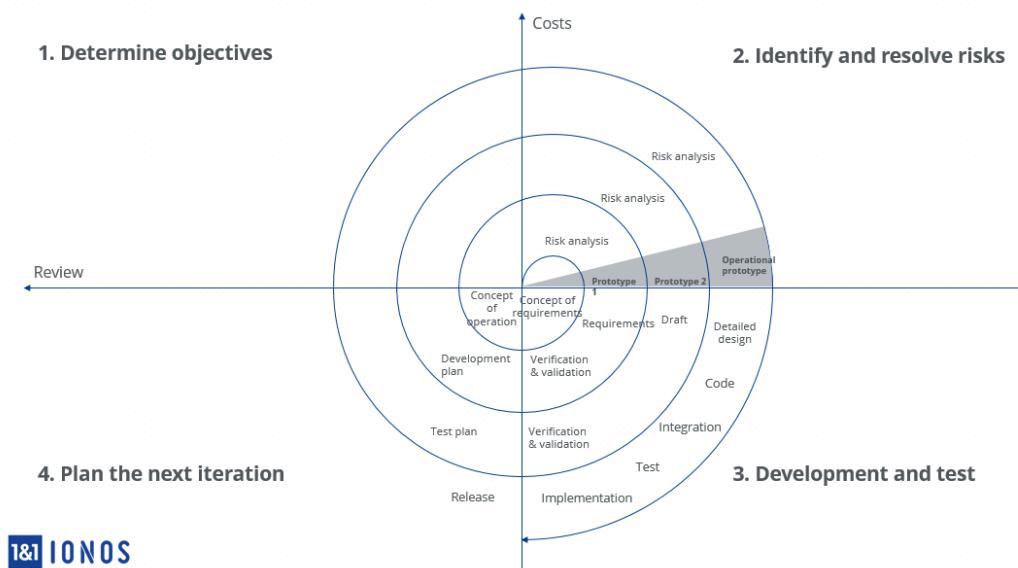


Figura 8: Modello a spirale

0.5 Progettazione architetturale

Una volta raggiunto un documento dei requisiti soddisfacente, ha inizio una nuova fase: la *progettazione architetturale* [?].

In questa fase si scomponete il sistema in *subsystems*, che poi verranno integrati per formare il sistema originale nella sua interezza. Questa scomposizione permette di concentrarsi su sottosistemi più semplici, riducendo quindi la complessità del problema.

Particolare attenzione in questa fase viene data al *subsystem control and communication*. Esso consiste nella definizione delle strutture di controllo (chi controlla chi) e delle *interfacce*⁴ dei singoli sottosistemi.

0.5.1 Architettura del sistema

Il nostro sistema si compone di quattro componenti:

- *Database*: si occupa di memorizzare i dati. Ha una piccola capacità di elaborazione: può ad esempio controllare che non ci siano duplicati o generare valori per nuovi inserimenti.
- *Sito web*: utilizzato dagli amministratori per la parte di *content management*⁵ e in parte dagli utenti per una migliore consultazione di alcuni dati (es. analisi dei dati).
- *Applicazione per smartphone*: utilizzato dagli utenti per interfacciarsi con il sistema.
- *API web*: si frappone tra database e i client, in modo da implementare la logica del sistema. È anche il principale controllore della sicurezza (effettua l'autenticazione, protegge le risorse).

L'API web comunica direttamente con il database utilizzando una connessione, sia essa *tcp* o *unix socket*. Le richieste vengono inviate dall'API al database utilizzando il formato accettato dal database (SQL).

⁴Un'*interfaccia* è una definizione del "modo di comunicare" che ha un sistema. Solitamente è rappresentato da un insieme di metodi con relativa signature (nel paradigma a oggetti)

⁵I *content management systems*, per quanto riguarda i siti web, servono per gestire le impostazioni e i contenuti dei siti web lato amministrativo.

L'API web fornisce i propri servizi tramite protocollo *REST*. I dati vengono codificati in formato *JSON*.

L'app invia tutte le richieste all'API web. Non può dialogare direttamente con il database.

Il browser ottiene le pagine web da un server web (statiche). All'interno delle pagine sono presenti degli *script*⁶ che richiedono e inviano i dati all'API web.

Di seguito un *diagramma di contesto*⁷ [?] che mostra i flussi di comunicazione e controllo tra i sottosistemi.

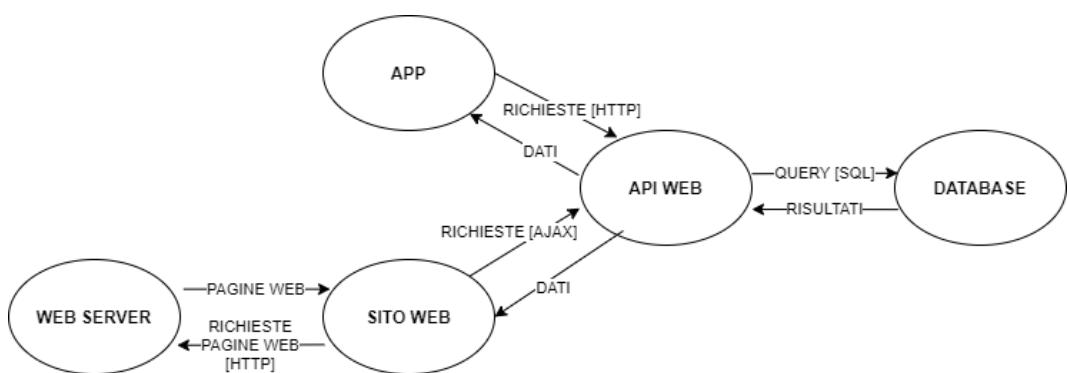


Figura 9: Diagramma di contesto del sistema

Approfondimento: Architettura REST e formato JSON

Il protocollo *REST* [?] (Representational state transfer) è uno tra i più usati nella costruzione di *web services*.

Esso sfrutta il protocollo *HTTP*. Si basa sul concetto di *risorsa*, identificata mediante un *URI*, e di operazione, identificata da un tipo. Ad esempio, una richiesta "*DELETE /persona/1234*" richiede la rimozione della persona con identificativo "1234".

⁶Gli *script* sono piccoli programmi eseguibili, solitamente scritti in linguaggio JavaScript, che vengono eseguiti all'interno del browser e permettono la creazione di pagine web dinamiche.

⁷I *Data Flow Diagrams* sono particolari diagrammi, molto semplici ed intuitivi, utilizzati per spiegare con un approccio strutturato e funzionale quelli che sono i processi e i flussi di dati tra di essi. In particolare i DFD di livello 0 sono anche chiamati DFD di contesto.

I dati, se presenti, vengono scritti nel *body* HTTP (richiesta o risposta). Solitamente vengono formattati in un formato standard: HTML, JSON oppure XML.

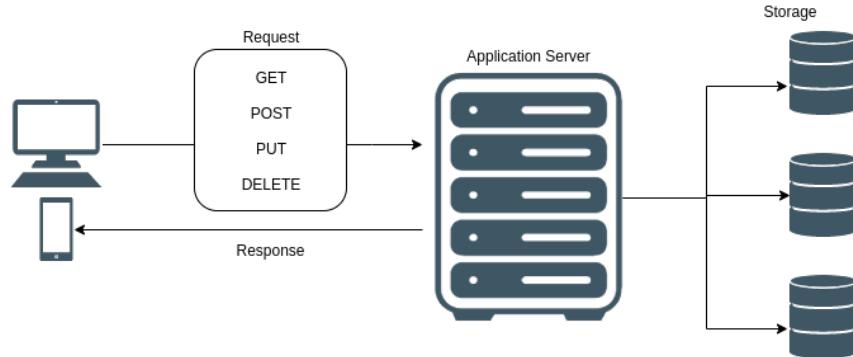


Figura 10: Struttura dell'architettura REST

Il formato *JSON* [?] (JavaScript Object Notation) è uno tra i più usati per l'encoding di dati non primitivi. È estremamente semplice da leggere e scrivere, sia per l'umano che per un software. È molto più compatto di XML, rendendolo preferibile per l'utilizzo umano. È il formato usato per rappresentare gli oggetti nel linguaggio *JavaScript*, e per questo risulta particolarmente familiare agli sviluppatori web.

```
{
  "users": [
    {
      "firstName": "Ray",
      "lastName": "Villalobos",
      "joined": {
        "month": "January",
        "day": 12,
        "year": 2012
      }
    },
    {
      "firstName": "John",
      "lastName": "Jones",
      "joined": {
        "month": "April",
        "day": 28,
        "year": 2010
      }
    }
  ]
}
```

Figura 11: Esempio di file json

0.5.2 Metodologia di lavoro

Una volta definita l'architettura del sistema, è necessario stabilire in che ordine e modo affrontare la progettazione e sviluppo dei singoli subsystems.

Dato che i subsystems individuati sono altamente indipendenti, il loro sviluppo potrebbe procedere in parallelo. Non avendo però necessità di terminare il progetto in tempi ristretti, risulta conveniente sviluppare prima il database, poi l'API, poi concorrentemente l'app e il website. In questo modo, è possibile evitare di sviluppare parte degli *stub*⁸, in quanto il modulo chiamato è già (in buona parte) implementato.

Lo sviluppo verrà effettuato in modo incrementale. Il primo incremento sarà di grosse dimensioni, in modo da includere tutto quanto richiesto dal committente. Incrementi successivi verranno usati per effettuare le modifiche necessarie, individuate durante l'attività di validazione oppure non strettamente necessarie per avere un prototipo funzionante (ad esempio, la cifratura dei dati).

Il committente verrà costantemente tenuto aggiornato rispetto agli sviluppi del progetto. Ad ogni iterazione del ciclo di sviluppo, gli verrà mostrato quanto prodotto per procedere alla validazione. I suoi commenti verranno utilizzati come base per la pianificazione dell'incremento successivo.

0.5.3 Stima dei tempi, costi e risorse

Avendo a disposizione molte più informazioni rispetto alla fase precedente, è possibile effettuare una migliore stima di tempi, costi e risorse.

La stima è la seguente:

- *Tempi*: 2 mesi per il database, 1 mese per l'API e 3 mesi per app+website, per quanto riguarda il primo e maggiore incremento.
- *Risorse*: I dispositivi propri del gruppo di sviluppo saranno sufficienti, almeno per quanto riguarda lo sviluppo. Per la validazione, potrà essere usato un qualche servizio di hosting cloud.

⁸Gli *stub* sono porzioni di codice che simulano la chiamata ad un modulo di livello inferiore. Sono utilizzati per il testing quando il modulo di livello inferiore non è ancora stato sviluppato.

- *Costi:* Per lo sviluppo, l'unico costo è quello della manodopera da parte del gruppo di lavoro. Essendo un progetto sviluppato a titolo gratuito, non sono previsti compensi per gli sviluppatori. Per la validazione, è previsto un massimo di euro 200 per quanto riguarda il servizio di cloud hosting.

Questo aggiornamento è stato subito comunicato al committente, per ragioni di trasparenza.

0.6 Qualità del software

Una nota particolare merita approfondimento: la qualità del software [?]. Essa non ha una definizione universale, ma va specificata di volta in volta.

La qualità del software dipende dalle capacità degli sviluppatori e dalle caratteristiche del progetto. Per ottenere una maggiore qualità, risulta quindi necessario affidare il progetto a sviluppatori maggiormente specializzati e/o porre maggiore sforzo nello sviluppo. Il tutto si traduce in un aumento generale dei costi di sviluppo.

Inoltre, per garantire alcune caratteristiche minime (come la disponibilità), può essere necessario investire maggiori risorse finanziarie per l'acquisto di servizi di hosting.

Non per tutti i progetti è necessario raggiungere la miglior qualità possibile: mentre sistemi *life-critical*⁹ devono garantire la miglior qualità possibile, sistemi di uso comune possono sacrificare parte della qualità in modo da ridurre tempi e costi.

Questo sistema deve avere le seguenti *software characteristics*:

- *Availability*: buona. È auspicabile arrivare al 99%. Una mancanza di disponibilità temporanea è sgradevole, ma non ha conseguenze catastrofiche.
- *Reliability*: ottima. Quando il sistema funziona, è importante che lo faccia senza malfunzionamenti. Dati errati potrebbero portare gli utilizzatori all'errore in condizioni critiche. È fortemente preferibile che il sistema non sia disponibile, piuttosto che funzioni in modo errato.
- *Error tolerance*: ottima. Essendo un sistema utilizzato da persone non specializzate anche in condizioni di fretta, è importante ridurre al minimo la possibilità che un errore umano pregiudichi il funzionamento del sistema.
- *Recoverability*: ottima. In caso di malfunzionamento o errore umano, è importante conservare una copia dei dati integra.

⁹I sistemi *life-critical* sono quei sistemi da cui dipende la vita delle persone, come i sistemi medici o i sistemi di volo degli aerei.

- *Efficiency*: sufficiente. Il sistema deve rispondere in tempi ragionevoli, ma non è importante che sia particolarmente rapido.
- *Security*: ottima. Solo i membri autorizzati devono poter effettuare modifiche. Letture non autorizzate devono essere evitate, anche se il loro avvenimento non causa conseguenze particolarmente serie.
- *Suitability*: ottima. Essendo un sistema sviluppato per un committente specifico, è importante che offra le funzionalità per lui migliori.
- *Learnability*: buona. Deve poter essere compreso da tutti, anche chi non ha specifiche competenze informatiche, in modo da incoraggiarne l'utilizzo. Si tollera un tempo di apprendimento iniziale.
- *Operability*: ottima. Una volta che l'utente ha imparato ad utilizzare il sistema, deve poterlo utilizzare sul campo in modo veloce e senza introdurre stress.

Approfondimento: Qualità del software

Secondo la definizione *ISO*¹⁰:

La *software quality* è la totalità delle features e caratteristiche del prodotto software legate alla sua abilità di soddisfare necessità/bisogni/aspettative scritte o implicite.

La *ISO 9000 series* contiene le guideline riguardanti *quality management* e *quality assurance*. In particolare, *ISO 9000-3* si occupa dell'applicazione di *ISO 9001* per lo sviluppo, fornitura e manutenzione del software.

Il processo di *quality assurance* è il processo generale relativo all'organizzazione attuata per raggiungere i requisiti di qualità contenuti nella *quality policy*. Il processo di *quality control* è composto dalle tecniche specifiche e attività svolte nel *quality system* per soddisfare i *quality requirements* del singolo progetto. In particolare, le *quality review* sono le azioni specifiche che valutano uno o più *quality requirement*.

La *ISO 9000 certification* viene assegnata da particolari commissioni di valutazione alle società che dimostrano di comprendere ed applicare le norme ISO nel loro operato. Essa è una delle certificazioni più importanti e di valore per dimostrare ai propri clienti di applicare elevati standard di qualità.

¹⁰L'*International Organization for Standardization* è un ente che si propone di formulare standard per l'utilizzo mondiale sulle materie più disparate.

Le *software quality characteristics* sono gli attributi di un software product la cui qualità è descritta e valutata.

Una *software quality metric* specifica la scala quantitativa e il metodo utilizzati per determinare oggettivamente il valore di una software quality characteristic.

Un *software quality model* è un insieme di software quality characteristics, che viene utilizzato per valutare la qualità di un software. Esso dipende fortemente dalle caratteristiche del software e del contesto d'uso.

ISO/IEC WD 9126-1 elenca e definisce in modo completo le caratteristiche applicabili ad un software quality model. Esse sono divise in tre grandi gruppi:

- *External characteristics*: valutate durante l'esecuzione.
- *Internal characteristics*: valutate mediante *software inspections*.
- *In use characteristics*: effetti dell'uso del software in un contesto specifico.



Figura 12: Caratteristiche della qualità

L'adozione di questi standard, anche senza disporre della certificazione, permette di avere strumenti oggettivi con cui valutare il proprio operato e poter fornire un prodotto di ottima qualità.

Approfondimento: Testing

Per valutare le caratteristiche della qualità, nonché per verificare il corretto funzionamento del sistema, lo strumento principale utilizzato è il testing [?].

Esso permette di *verificare* e *valutare*¹¹ il sistema durante tutte le fasi, dalla progettazione alla consegna.

¹¹La *verifica* è il processo che compara il software con le specifiche, in modo da individuare errori di programmazione e controllare che il software faccia *quanto stabilito* ("stiamo costruendo bene il sistema richiesto?"). La *validazione* invece controlla che il software incontri davvero le *necessità del committente*, anche se esse dovessero discostarsi dai requisiti ("stiamo costruito il sistema giusto per il committente?").

La principale distinzione tra tipologie di testing è tra *software inspections* e *verifiche in esecuzione*. Le prime consistono nell'esaminazione diretta del codice da parte di un programmatore, e quindi possono essere eseguite durante tutto il ciclo di vita. Le verifiche in esecuzione sono eseguite valutando la risposta del sistema a particolari input, e quindi necessitano di avere un eseguibile funzionante.

Il *component testing* si occupa del testing di un singolo componente, o sottosistema. L'*integration testing* testa invece le interazioni tra i componenti, mettendoli assieme. Affinché l'integration testing abbia successo, è necessario prima aver verificato il funzionamento di tutti i componenti.

Il *black-box testing* inventa i casi di test guardando solamente le specifiche. I *testcase*¹² In genere si controllano alcuni input "normali" e alcuni input "anormali". Particolare attenzione viene data ai *corner case*.

Il *white-box testing* invece deriva i testcase dal codice. Si tenta di eseguire ogni linea di codice, dando particolare attenzione alle strutture di controllo (come condizioni e cicli). A riguardo, il *path testing* studia i testcase in modo da eseguire una volta tutti i possibili cammini nel codice.

L'*interface testing* prova a dialogare con un componente tramite le sue interfacce. Si dà importanza all'utilizzo in modo corretto delle interfacce (*interface misusing*), alle assunzioni che il chiamante fa riguardo all'interfaccia (*interface misunderstanding*) e all'interazione tra componenti con velocità diverse (*timing errors*).

La tecnica del *regression testing* consiste nell'effettuare, ad ogni incremento, tutti i test precedenti, per verificare di non aver introdotto difetti.

Lo *stress test* carica il sistema o una parte oltre il suo limite in modo da verificarne il funzionamento e il degrado delle performance.

Per effettuare i test spesso è necessario utilizzare degli *stub* o *test driver*. Gli stub sono parti di codice che simulano moduli di livello inferiore (chiamati), mentre i test driver simulano moduli di livello superiore (chiamanti).

¹²I *testcase* sono coppie, in cui ad un input viene associato il comportamento del sistema che ci si aspetta con quell'input.

Il *validation testing* viene usato al termine dello sviluppo per validare il software, e decidere se sono necessarie modifiche o si può consegnare. Il modo migliore per svolgerlo è far provare il sistema da quelli che saranno gli utilizzatori finali per un congruo lasso di tempo e senza dare indicazioni (perché poi nell'effettivo utilizzo non saranno seguiti dal vivo).

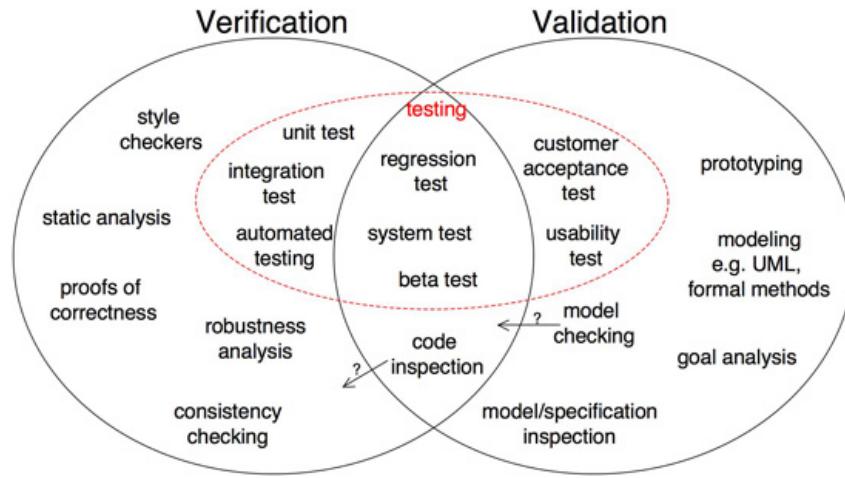


Figura 13: Verifica e validazione

0.7 Database

In questa sezione verrà presentato lo sviluppo del database [?], prima azione compiuta a seguito della progettazione architetturale. Tutti gli altri sottosistemi dipendono da esso, quindi durante la progettazione è stato necessario tenere in considerazione le esigenze di essi.

0.7.1 Progettazione concettuale

Il primo passaggio nella progettazione di una base di dati è la *progettazione concettuale*. Essa si basa sui requisiti e produce una prima rappresentazione del contenuto della base di dati, svincolata dalla sua effettiva implementazione. Il suo risultato è lo *schema Entità-Relazioni*.

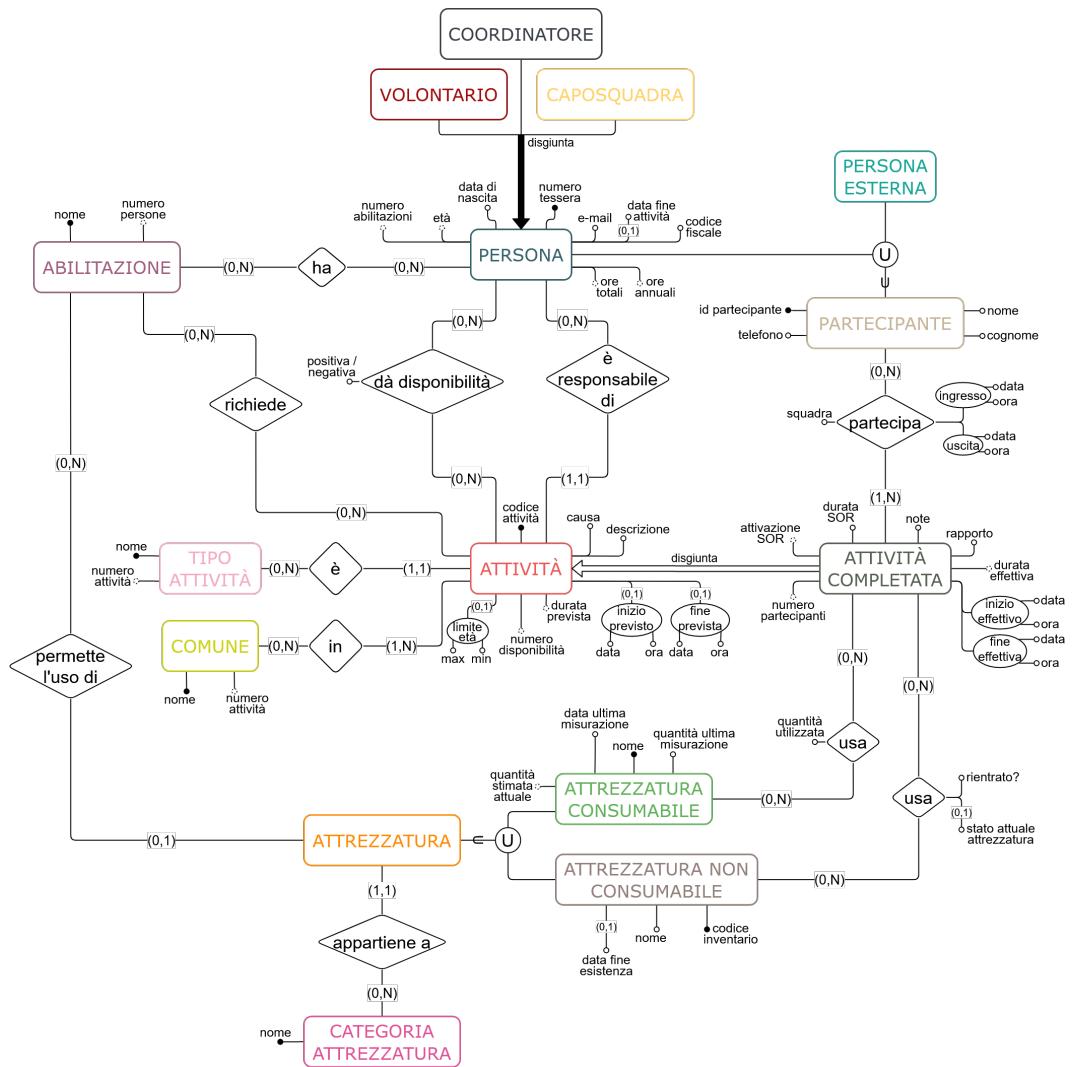


Figura 14: Schema ER

Per modellare la base di dati, si è scelto di utilizzare il modello *relazionale*, in quanto la base di dati è fortemente strutturata.

0.7.2 Progettazione logica

Tutte le relazioni di tipo N a N , non essendo direttamente rappresentabili nel modello relazionale, sono state reificate introducendo adeguate entità intermedie. Al termine di ciò, tutte le relazioni presenti sono di tipologia 1 a N oppure 1 a 1 .

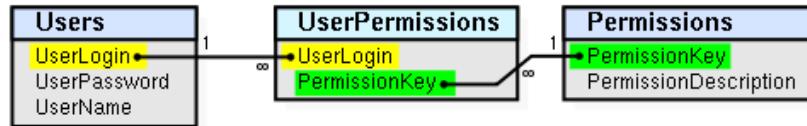


Figura 15: Esempio di tabella associativa usata per modellare una relazione N a N

A seguito è stata eseguita l'*analisi delle ridondanze*. Essa prende in input lo schema ER, la *tabella delle frequenze*¹³ e la *tabella dei volumi*¹⁴ per produrre un insieme di *tabelle degli accessi*. Esse permettono, per ogni ridondanza, di scegliere se risulti computazionalmente conveniente mantenerla o eliminarla.

Operazione	Tipo ¹	Frequenza [volte/anno]
Data un'attività a e una persona p , indicazione della disponibilità di p per a	I	9000
Data un'attività a , calcolo del numero di disponibilità per a	I	1000
Data un'attività completata a e una persona p , indicazione della partecipazione di p ad a	I	500
Data una persona p , calcolo delle ore svolte da p nell'ultimo anno solare	I	300
Data una persona p , calcolo del totale ore malta da p	I	100

Figura 16: Frammento della tabella delle frequenze

¹³La *tabella delle frequenze* stima la frequenza di esecuzione delle principali operazioni sulla base di dati.

¹⁴La *tabella dei volumi* stima il numero di istanze per ciascuna entità e ciascuna relazione presenti nella base di dati.

Label ²	Nome	Tipo ³	Nr. istanze
$\langle B \rangle$	Abilitazione	E	30
$\langle A \rangle$	Attività	E	1000
$\langle AC \rangle$	Attività completata	E	1000
$\langle T \rangle$	Attrezzatura	E	40
$\langle TC \rangle$	Attrezzatura consumabile	E	10
$\langle TN \rangle$	Attrezzatura non consumabile	E	30
$\langle TT \rangle$	Categoria attrezzatura	E	5
$\langle C \rangle$	Comune	E	300
$\langle PP \rangle$	Partecipante	E	150
$\langle P \rangle$	Persona	E	100
$\langle PE \rangle$	Persona esterna	E	50
$\langle AT \rangle$	Tipo attività	E	10
$\langle B:T \rangle$	$\langle B \rangle$ permette l'uso di $\langle T \rangle$	R	15
$\langle A:AT \rangle$	$\langle A \rangle$ è $\langle AT \rangle$	R	1000
$\langle A:B \rangle$	$\langle A \rangle$ richiede $\langle B \rangle$	R	1000

Figura 17: Frammento della tabella dei volumi

Ore annuali/ore totali Si considerano gli attributi derivati *ore annuali* e *ore totali* relativi all'entità *persona*. Si può in questo caso effettuare un'analisi complessiva poiché entrambi questi attributi riguardano le stesse operazioni.

Con attributi derivati

- Data un'attività completata a e una persona p , indicazione della partecipazione di p ad a .

Nome	Tipo	Accessi	R/W
$\langle P \rangle$	E	1	W
$\langle PP:AC \rangle$	R	1	W

- Data una persona p , calcolo delle ore svolte da p nell'ultimo anno solare.

Nome	Tipo	Accessi	R/W
$\langle P \rangle$	E	1	R

- Data una persona p , calcolo del totale ore svolte da p .

Nome	Tipo	Accessi	R/W
$\langle P \rangle$	E	1	R

Senza attributi derivati

- Data un'attività completata a e una persona p , indicazione della partecipazione di p ad a .

Nome	Tipo	Accessi	R/W
$\langle PP:AC \rangle$	R	1	W

- Data una persona p , calcolo delle ore svolte da p nell'ultimo anno solare.

Nome	Tipo	Accessi	R/W
$\langle PP:AC \rangle$	R	5	R

- Data una persona p , calcolo del totale ore svolte da p .

Nome	Tipo	Accessi	R/W
$\langle PP:AC \rangle$	R	50	R

✓ Con ridondanza: $4 \cdot 500 + 1 \cdot 300 + 1 \cdot 100 = 2400$
 ✗ Senza ridondanza: $2 \cdot 500 + 5 \cdot 300 + 50 \cdot 100 = 7500$

Figura 18: Esempio di tabella degli accessi per l'analisi delle ridondanze

Il risultato complessivo è lo *schema ER ristrutturato*.

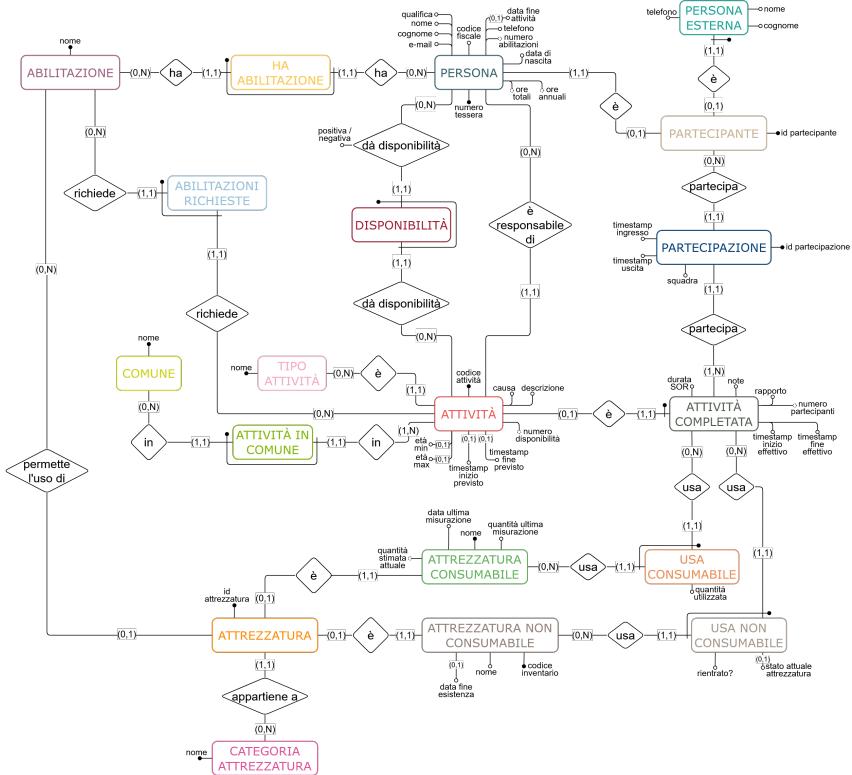


Figura 19: Schema ER ristrutturato

0.7.3 Progettazione fisica e implementazione

Volendo sfruttare l'occasione del corso di laboratorio di Basi di Bati, a cui tutti i membri del gruppo di lavoro fanno parte, abbiamo deciso di procedere all'implementazione con il database relazionale *PostgreSQL* [?].

L'implementazione delle entità e relazioni risulta naturale traducendo direttamente in SQL il contenuto dello schema ER ristrutturato.

Listing 1: Esempio di implementazione di entità e relazioni

```

1 CREATE TYPE qualifica_persona AS ENUM (
2   'volontario', 'caposquadra', 'coordinatore',
3 );
4
5 CREATE TABLE persona (
6   numero_tessera text PRIMARY KEY, — Text = varchar(inf)
7   id_partecipante int NOT NULL, — Come chiave esterna , un
     serial diventa int | Il vincolo UNIQUE e' controllato dal
     trigger apposito

```

```

8     codice_fiscale text NOT NULL UNIQUE, — Era una chiave
9         candidata
10    nome text NOT NULL,
11    cognome text NOT NULL,
12    email text NOT NULL,
13    telefono text NOT NULL,
14    data_di_nascita date NOT NULL,
15    data_fine_attivita date DEFAULT NULL,
16    qualifica qualifica_persona NOT NULL,
17    numero_abilitazioni int NOT NULL DEFAULT 0,
18    ore_annuali interval NOT NULL DEFAULT interval '0 days',
19    ore_totali interval NOT NULL DEFAULT interval '0 days',
20    FOREIGN KEY (id_partecipante) REFERENCES partecipante(
21        id_partecipante) ON DELETE NO ACTION ON UPDATE CASCADE
22 );
23
24 CREATE TABLE persona_esterna (
25     id_partecipante int PRIMARY KEY, — Il vincolo UNIQUE e'
26         controllato dal trigger apposito
27     nome text NOT NULL,
28     cognome text NOT NULL,
29     telefono text NOT NULL,
30     FOREIGN KEY (id_partecipante) REFERENCES partecipante(
31         id_partecipante) ON DELETE NO ACTION ON UPDATE CASCADE
32 );
33
34 CREATE TABLE ha_abilitazione (
35     numero_tessera text ,
36     nome_abilitazione text ,
37     PRIMARY KEY (numero_tessera , nome_abilitazione) ,
38     FOREIGN KEY (numero_tessera) REFERENCES persona(numero_tessera
39         ) ON DELETE CASCADE ON UPDATE CASCADE,
40     FOREIGN KEY (nome_abilitazione) REFERENCES abilitazione(
41         nome_abilitazione) ON DELETE CASCADE ON UPDATE CASCADE
42 );

```

Sono stati definiti gli indici sui campi maggiormente utilizzati dalle query per la ricerca.

Listing 2: Esempio di implementazione di indici

```

1 CREATE INDEX idx_attivita_inizio ON attivita(
2     timestamp_inizio_previsto , timestamp_fine_previsto);
3 CREATE INDEX idx_attivita_completata_inizio ON attivita_completata
4     (timestamp_inizio_effettivo , timestamp_fine_effettivo); — Per
5     quantita' stimata di attrezzatura consumabile
6 CREATE INDEX idx_persona_partecipante ON persona(id_partecipante);

```

Mediante la codifica di opportuni *trigger*¹⁵ in linguaggio *plpgsql*, abbiamo realizzato i principali controlli di integrità non verificabili dal dbms tramite i soli *constraints*. Inoltre, trigger aggiuntivi sono stati aggiunti per facilitare l'inserimento di alcuni dati.

Listing 3: Esempio di implementazione di trigger

```

1 — Vincolo di esclusività 'attrezzatura' (no duplicati) in
   inserimento
2 CREATE OR REPLACE FUNCTION check_id_attrezzatura()
3 RETURNS TRIGGER
4 LANGUAGE plpgsql AS $$ 
5 BEGIN
6     IF EXISTS (
7         (SELECT id_attrezzatura FROM attrezzatura_consumabile
8          WHERE id_attrezzatura = new.id_attrezzatura)
9         UNION
10        (SELECT id_attrezzatura FROM attrezzatura_non_consumabile
11          WHERE id_attrezzatura = new.id_attrezzatura)
12    )
13    THEN Raise Exception 'Duplicate id_attrezzatura: %', new.
14      id_attrezzatura;
15  END IF;
16  RETURN new;
17 END;
18 $$;
19
20
21 DROP TRIGGER IF EXISTS trigger_02_insert_consumabile ON
22   attrezzatura_consumabile; — Il '02' nel nome serve perché i
23   trigger su una tabella sono controllati in ordine alfabetico
24 CREATE TRIGGER trigger_02_insert_consumabile
25   BEFORE INSERT ON attrezzatura_consumabile
26   FOR EACH ROW
27   EXECUTE PROCEDURE check_id_attrezzatura();
28
29
30 DROP TRIGGER IF EXISTS trigger_insert_non_consumabile ON
31   attrezzatura_non_consumabile;
32 CREATE TRIGGER trigger_insert_non_consumabile
33   BEFORE INSERT ON attrezzatura_non_consumabile
34   FOR EACH ROW
35   EXECUTE PROCEDURE check_id_attrezzatura();
```

¹⁵I *trigger* sono particolari script eseguibili all'interno di un dbms, a seguito della modifica dei suoi contenuti.

0.7.4 Deploy locale

Per effettuare il testing preliminare, è stato effettuato il deploy di un’istanza locale tramite docker.

L’immagine scelta è *postgres*, a cui basta passare le credenziali di amministrazione tramite environment.

Listing 4: Configurazione di docker compose per postgres

```

1  postgres:
2    container_name: protezionecivile_pg
3    image: postgres
4    environment:
5      POSTGRES_USER: postgres
6      POSTGRES_PASSWORD: postgres
7      PGDATA: /data/postgres
8    volumes:
9      - pgdata:/data/postgres
10     - ./database:/base
11    ports:
12      - '5432:5432'

```

Approfondimento: Utilizzo di Docker

Docker [?] è una piattaforma software che permette di creare, testare e distribuire applicazioni in modo semplice e sicuro.¹⁶

Esso raccoglie il software in unità standardizzate chiamate *container*, le quali offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime.

Lavorando in un ambiente virtualizzato, si ha a disposizione un intero sistema operativo comprensivo di memoria stabile e risorse hardware (scheda di rete, gpu, ecc...), senza però rischiare di compromettere il proprio dispositivo. Infatti, se qualcosa dovesse andare storto, basta eliminare il container difettoso ed avviare uno nuovo!

Il funzionamento è standardizzato: lo stesso container può essere eseguito senza sforzo in qualunque macchina che abbia installato Docker server. Un

¹⁶<https://aws.amazon.com/it/docker/>

servizio testato in ambiente locale molto probabilmente funzionerà senza problemi una volta caricato nell'ambiente di produzione. Per questo, docker è particolarmente indicato per l'utilizzo nel cloud.

Docker è anche molto utile per gestire la scalabilità: con semplici comandi è possibile lanciare nuove istanze per affrontare immediatamente rapidi picchi di carico o fornire ridondanza.

Docker compose è un'estensione di Docker che permette di specificare, mediante un file di configurazione unico in formato yaml, un insieme di istanze che lavorano in modo sinergico. In questo modo, è possibile usare un comando unico per effettuare il deploy di tutto il necessario per il progetto.

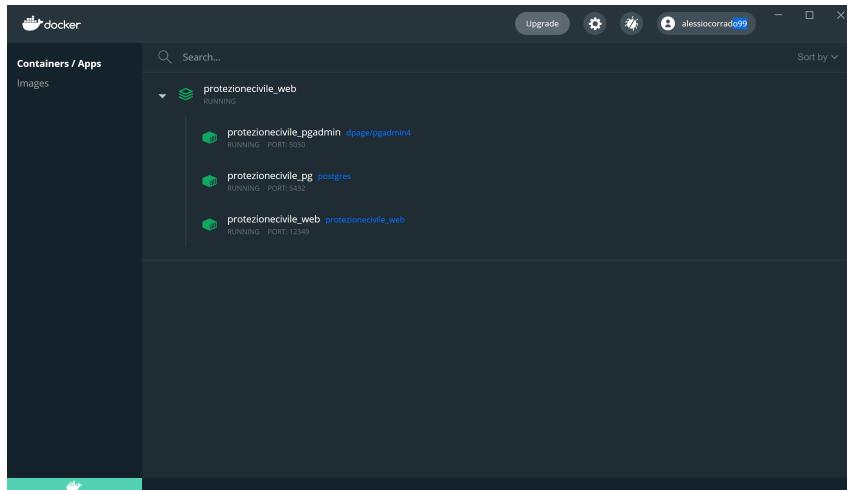


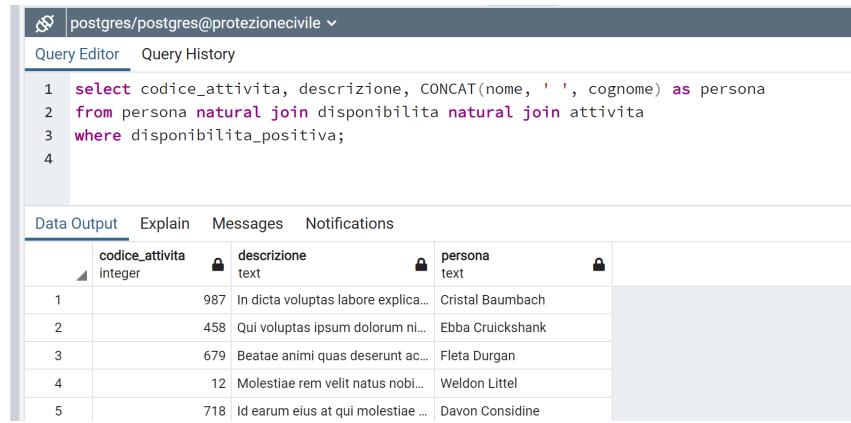
Figura 20: Utilizzo di Docker Desktop con i container usati nel progetto

0.7.5 Testing

Per interfacciarsi con il postgres, si è scelto di usare la GUI *pgAdmin4* [?].

Un primo testing è stato effettuato inserendo manualmente query in modo da popolare il database con pochi dati, effettuare ricerche e modificarne il contenuto. Per ogni vincolo di integrità, è stata eseguita almeno una query che andasse a verificare il comportamento.

Successivamente, tramite l'utilizzo della libreria *Faker* per linguaggio PHP, abbiamo popolato il database con dati realistici e conformi con la tabella dei volumi.



The screenshot shows the pgAdmin4 interface. At the top, there's a header bar with a connection icon and the text "postgres/postgres@protezionecivile". Below it is a navigation bar with tabs: "Query Editor" (which is active), "Query History", "Data Output", "Explain", "Messages", and "Notifications". The main area contains a SQL query and its execution results.

```
1 select codice_attivita, descrizione, CONCAT(nome, ' ', cognome) as persona
2 from persona natural join disponibilita natural join attivita
3 where disponibilita_positiva;
4
```

	codice_attivita integer	descrizione text	persona text
1	987	In dicta voluptas labore explica...	Cristal Baumbach
2	458	Qui voluptas ipsum dolorum ni...	Ebba Cruickshank
3	679	Beatae animi quas deserunt ac...	Fleta Durgan
4	12	Molestiae rem velit natus nobi...	Weldon Littel
5	718	Id earum eius at qui molestiae ...	Davon Considine

Figura 21: Attività di testing sul database con pgAdmin4

0.8 API web

Una volta concluso lo sviluppo del database, è stato affrontata la progettazione e sviluppo dell'API web [?].

0.8.1 Progettazione

Per lo sviluppo di questa parte è stato scelto un approccio di tipo incrementale. Il primo incremento contiene solamente il minimo indispensabile per eseguire un server. Ad ogni incremento successivo, vengono aggiunti e testati nuovi endpoint.

Per avere una migliore modularizzazione del codice, viene inizialmente costruita una libreria di supporto. Essa permette di effettuare in modo agevole le operazioni più comuni, come l'accesso al database e la formattazione della risposta.

La cache, implementata con *Redis* [?], permette di velocizzare le operazioni di recupero dei dati in lettura. Infatti, con essa non è necessario che il database e il server API ricalcolino la risposta ad ogni chiamata, ma possono inviare direttamente la risposta, se presente in cache.

Per l'implementazione, si è scelto di usare il linguaggio *Python 3* [?] tramite framework *Flask* [?]. Esso dispone nativamente di un server di sviluppo.

Approfondimento: Utilizzo di git per lo sviluppo incrementale

I sistemi di *version control* sono sistemi specializzati nella gestione dei cambiamenti dei file. Uno dei sistemi di version control maggiormente utilizzati nell'ambito della produzione e manutenzione di software è il *Git* [?].

Il contenuto di un file in un certo istante viene chiamato *versione*. Ad ogni versione è associato un *timestamp*, e optionalmente un identificativo. L'identificativo spesso è composto da un nome o da un codice.

Solitamente, si parte con un progetto vuoto. Alla creazione di ciascun documento, si ha la creazione della sua *versione iniziale*. Ogni volta che si modifica un documento, ne viene creata una nuova versione, che va a sostituire la precedente.

Al raggiungimento di una *milestone*, risoluzione di un bug, implementazione di una feature o qualsiasi altro punto di avanzamento dei lavori ritenuto "di interesse", tutte le ultime versioni dei documenti interessati vengono contrassegnate con un messaggio. Questa procedura si chiama *commit*.

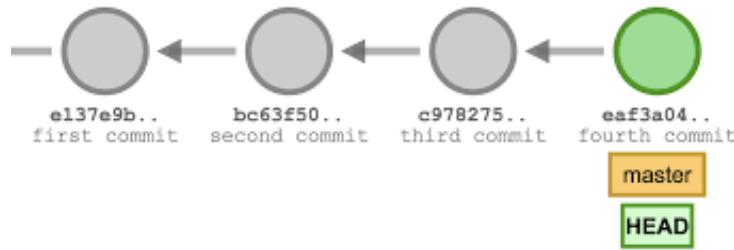


Figura 22: Esempio di sequenza di commit, ciascuno con il proprio identificativo

Nel caso in cui qualcosa dovesse andare storto, le operazioni di *reset* e *checkout* permettono di ripristinare la situazione a subito dopo uno qualsiasi dei commit precedenti. Il messaggio associato a ciascun commit permette di identificare facilmente il punto corretto da cui riprendere i lavori.

Nel caso si stia lavorando ad un'estensione, oppure si voglia lavorare contemporaneamente a più incrementi, la strategia ottimale consiste nel creare *branch multipli*. Il branch *master* contiene la sequenza di commit del "programma principale". In qualsiasi momento, è possibile creare un nuovo branch e su di esso proseguire con il lavoro.

L'operazione di *merge*, solitamente effettuata tra il *branch di lavoro* e *master*, permette di riunire i due branch, fondendoli. Il merge può consistere in una sovrascrittura di un branch sull'altro, oppure in una fusione in cui si tengono alcuni dati di un branch e alcuni dell'altro (nel caso in cui siano stati effettuati commit su entrambi i branch dopo la divisione).

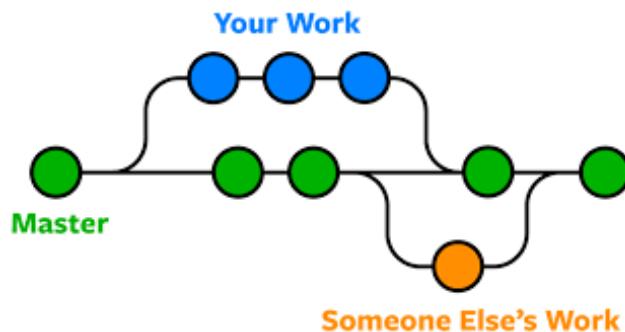


Figura 23: Esempio dell’uso di git in un progetto condiviso

Le operazioni di *push* e *pull* permettono di sincronizzare il contenuto di un’istanza locale di git con un server condiviso. Ciò risulta fondamentale nel momento in cui si lavora in team, permettendo ad ogni membro di lavorare sui suoi branch e poi caricarli o fonderli assieme in uno spazio comune.

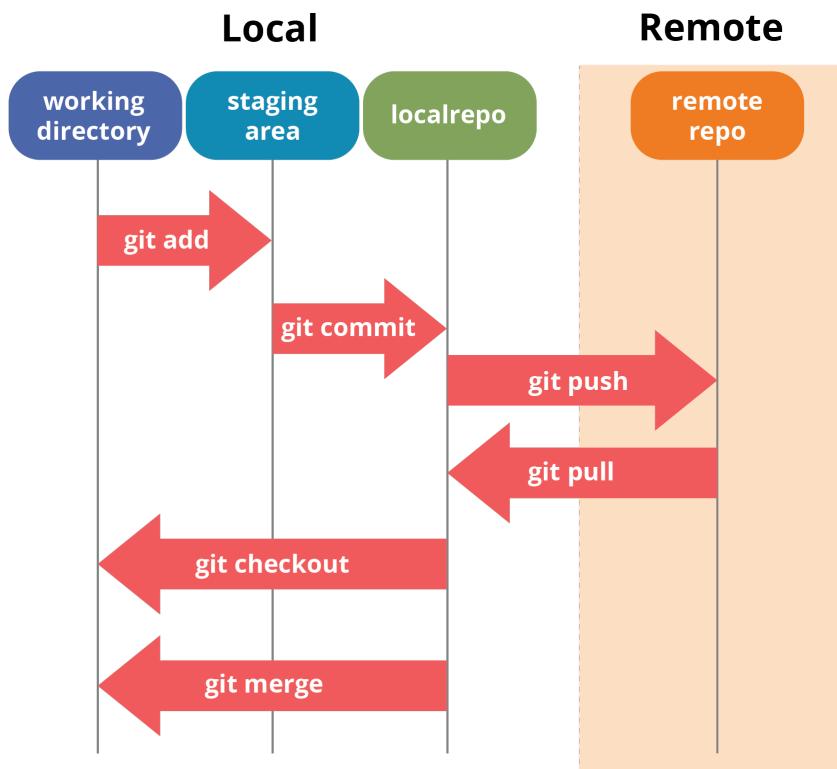


Figura 24: Operazioni principali durante l’utilizzo di git

L'utilizzo di git si sposa con la metodologia dello *sviluppo incrementale*. Infatti, è naturale associare ad ogni incremento un commit. Inoltre, l'utilizzo di branch separati permette di sviluppare in parallelo incrementi diversi senza che essi interferiscano fra loro, e permette di tenere separata la versione di *produzione* da quella di *sviluppo*.

Approfondimento: WSGI

La *Web Server Gateway Interface (WSGI)* [?] è una calling convention seguita dai server web per inoltrare le richieste ad applicazioni web o framework scritti in Python.

In precedenza, i webserver si interfacciavano con le webapp utilizzando una grande varietà di protocolli, a volte persino sviluppati ad hoc. L'adozione dell'interfaccia comune WSGI promuove la scrittura di codice portabile, riutilizzabile e standardizzato.

WSGI ha due lati:

- *server/gateway*: un webserver completo, come *nginx* o *apache*, oppure una lightweight app che comunica con un server;
- *application/framework*: un eseguibile scritto in Python che viene richiamato per gestire le richieste.

I *middleware* sono componenti inseriti tra server e application, che quindi implementano entrambi i lati di WSGI. Essi possono elaborare le richieste e/o inoltrarle ad altre applicazioni WSGI. Un utilizzo comune dei middleware è per inserire layer di log, autenticazione o gestione degli errori.

Un'applicazione compatibile con wsgi è composta da particolari funzioni generatrici, che accettano due parametri, *environ* e *start_response*, e ritornano un flusso di byte (la risposta). Di seguito è riportato un esempio.

Listing 5: Applicazione hello world

```

1 #application: il nome della funzione
2 #environ: un dizionario contenente le variabili d'ambiente CGI, i
           parametri della richiesta e altri parametri user-defined
3 #start_response: una funzione che accetta i parametri (status,
                  response_headers)
4 def application(environ, start_response):
5     start_response('200 OK', [('Content-Type', 'text/plain')])
6     yield b'Hello, World!\n'

```

Di seguito invece un'implementazione dimostrativa lato chiamante.

Listing 6: Implementazione dimostrativa chiamante

```

1 from io import BytesIO
2
3 def call_application(app, environ):
4     status = None
5     headers = None
6     body = BytesIO()
7
8     def start_response(rstatus, rheaders):
9         nonlocal status, headers
10        status, headers = rstatus, rheaders
11
12    app_iter = app(environ, start_response)
13    try:
14        for data in app_iter:
15            assert status is not None and headers is not None, \
16                "start_response() was not called"
17            body.write(data)
18    finally:
19        if hasattr(app_iter, 'close'):
20            app_iter.close()
21    return status, headers, body.getvalue()
22
23 environ = {...}
24 status, headers, body = call_application(app, environ)

```

Il framework Flask implementa lo standard WSGI. La sua facilità di utilizzo, unita all'elevata versatilità e alla compattezza del codice, lo rende particolarmente adatto allo sviluppo web.

Listing 7: Hello world con Flask

```

1 app = Flask(__name__)
2
3 @app.route('/')
4 def hello():
5     return "Hello World!"

```

0.8.2 Inizializzazione del contesto

Inanzitutto è stato scritto il codice relativo all'inizializzazione di Flask e alla connessione con database e cache.

Listing 8: Inizializzazione del server

```

1 app = Flask(__name__, template_folder="template")
2
3 if __name__ == '__main__':
4     app.run(host='0.0.0.0', port=8080, debug=True)

```

Listing 9: Connessione al database

```

1 db_config = {
2     'user': os.environ.get('SQL_USERNAME'),
3     'password': os.environ.get('SQL_PASSWORD'),
4     'database': os.environ.get('SQL_DATABASE_NAME'),
5     'host': os.environ.get('SQL_HOST')
6 }
7 cnxpool = psycopg2.pool.ThreadedConnectionPool(minconn=1, maxconn
     =100, **db_config)

```

Listing 10: Connessione alla cache Redis

```

1 redis_host = os.environ.get('REDISHOST')
2 redis_port = os.environ.get('REDISPORT')
3
4 redis_client = redis.StrictRedis(host=redis_host, port=int(
    redis_port)) if redis_host is not None else None

```

0.8.3 Costruzione libreria di supporto

La libreria di supporto è stata costruita in modo da supportare al meglio lo sviluppo di questo progetto. Essendo però generica, potrà essere facilmente utilizzata dagli sviluppatori per eventuali progetti futuri.

La funzione *do* è quella di più basso livello. Essa inanzitutto apre una connessione al database (*cnx*). Viene poi applicata la funzione *query*, contenente la logica della chiamata. Essa restituisce un risultato sotto forma di oggetto, il quale viene formattato per l'invio tramite *applyRes*.

Listing 11: Funzione *do*

```

1 def defaultToJson(data):
2     return json.dumps(data, indent=4, sort_keys=True, default=str)
3
4
5 def do(query, applyRes = defaultToJson):
6     cnx = cnxpool.getconn()
7     with cnx.cursor(cursor_factory=psycopg2.extras.RealDictCursor)
        as cursor:
8         res = query(cursor)

```

```
9      res = applyRes(res)
10     cnx.commit()
11     cnxpool.putconn(cnx)
12
13     return res
```

Le funzioni *cursor_** astraggono le operazioni più comuni sul database, fornendo un'interfaccia per effettuare le query standard di *insert*, *update* e *delete* su singola tabella. *select* non è inclusa in quanto utilizzare una funzione generica è più difficoltoso che scrivere direttamente la query in linguaggio SQL.

Listing 12: Funzioni *cursor_**

```
1 #data: dict contenente i dati da inserire/modificare
2 #table: nome tabella
3 #fields: lista degli attributi della tabella
4 #cond: lista degli attributi che compongono la chiave primaria
5 #returning: stringa contenente il nome da associare alla clausola
   returning (per la generazione di valori da parte del database)
6
7 def cursor_insert(cursor, data, table, fields, returning=None):
8     #inserisci solo valori effettivamente presenti in data
9     #cio' permette di inserire solo alcune colonne, e lasciare il
   valore di default nelle rimanenti
10    fields = list(filter(lambda x: x in data.keys(), fields))
11
12    #generazione dinamica della query
13    #'%s' viene usato come placeholder per iniettare i valori in
   modo safe (contro attacchi di tipo sql injection)
14    s1 = ", ".join(fields)
15    s2 = ", ".join(map(lambda x: "%s", fields))
16
17    query = "insert into " + table + "(" + s1 + ") values (" + s2
   + ")"
18
19    if returning is not None:
20        query += " returning " + returning
21
22    #binding tra placeholder e valori effettivi
23    vals = [data[i] for i in fields]
24
25    #esecuzione della query nel db
26    cursor.execute(query, vals)
27
28 def cursor_update(cursor, data, table, fields, cond, returning=
   None):
```

```

30     fields = list(filter(lambda x: x in data, fields))
31
32     s1 = ", ".join(map(lambda x: x + "=%" + str(x), fields))
33     s2 = ", ".join(map(lambda x: x + "=%" + str(x), cond))
34
35     query = "update " + table + " set " + s1 + " where " + s2
36     if returning is not None:
37         query += " returning " + returning
38
39     vals = [data[i] for i in fields + cond]
40
41     cursor.execute(query, vals)
42
43 def cursor_delete(cursor, data, table, cond):
44     s2 = " and ".join(map(lambda x: x + "=%" + str(x), cond))
45
46     query = "delete from " + table + " where " + s2
47
48     vals = [data[i] for i in cond]
49
50     cursor.execute(query, vals)

```

Le funzioni *generate_** creano gli endpoint associati a metodi standard, per effettuare le operazioni corrispondenti su singola tabella.

Gli endpoint sono standardizzati. Ad esempio, il metodo per effettuare un update dell'entità *persona* viene generato all'endpoint *POST /api/update_persona*.

Listing 13: Funzioni *generate_**

```

1 #table, fields, cond, returning hanno lo stesso significato delle
  funzioni cursor_*
2 #prefix: il prefisso delle chiamate API
3
4 def generate_create(table, fields, returning, prefix="/api"):
5     def f_create():
6         #contenuto del body della richiesta, parsing da formato
        json
7         data = request.json
8
9     def f(cursor):
10        #inserimento nel database tramite interfaccia cursor_*
11        cursor_insert(cursor, data, table, fields, returning)
12
13        #default per risposta con successo
14        res = {"message": "ok"} 

```

```
15
16      #includi nella risposta quanto generato dalla clausola
17      returning
18      if returning:
19          v = cursor.fetchone()
20          res[returning] = v[returning]
21
22      return res
23
24  return do(f)
25
26  #creazione dinamica dell'endpoint associato alla richiesta
27  app.add_url_rule(prefix + "/create_" + table, "create_" +
28      table, f_create, methods=["POST"])
29
30 def generate_update(table, fields, cond, returning, prefix="/api"):
31     def f_update():
32         data = request.json
33
34         def f(cursor):
35             cursor_update(cursor, data, table, fields, cond,
36                           returning)
37
38         ... come insert ...
39
40     app.add_url_rule(prefix + "/update_" + table, "update_" +
41                      table, f_update, methods=["POST"])
42
43 def generate_set(table, fields, prefix="/api"):
44     def f_set():
45         data = request.json
46
47         def f(cursor):
48             #elimina tutte le righe nel database
49             cursor.execute("delete from " + table)
50
51             #inserisce in modo batch ogni riga in input
52             for elem in data:
53                 cursor_insert(cursor, elem, table, fields)
54
55         return {"message": "ok"}
56
57     return do(f)
58
```

```

59     app.add_url_rule(prefix + "/set_" + table, "set_" + table,
60                       f_set, methods=["POST"])
61
62     def generate_delete(table, cond, prefix="/api"):
63         def f_delete():
64             data = request.json
65
66             def f(cursor):
67                 cursor_delete(cursor, data, table, cond)
68
69             return {"message": "ok"}
70
71         return do(f)
72
73     app.add_url_rule(prefix + "/delete_" + table, "delete_" +
74                     table, f_delete, methods=["DELETE"])
75
76     def generate_get(table, query=None, prefix="/api"):
77         #la query di default seleziona tutte le colonne e tutte le
78         #righe della tabella
79         if query is None:
80             query = "select * from " + table
81
82         def f_get():
83             data = request.json
84
85             def f(cursor):
86                 cursor.execute(query)
87
88                 #nel caso la query ritorni una sola riga, e' comunque
89                 #contenuta in una lista unitaria [row1]
90                 #lista vuota [] nel caso in cui la query non ritorni
91                 #alcuna riga
92                 return cursor.fetchall()
93
94             return do(f)
95
96     app.add_url_rule(prefix + "/get_" + table, "get_" + table,
97                      f_get, methods=["GET"])

```

La funzione *generate_crsd* genera gli endpoint per effettuare le operazioni *create*, *read*, *update* e *delete* su una singola entità.

Listing 14: generate_crsd

```

1 def generate_crsd(table, fields, cond=None, returning=None, prefix
2     ="/api"):
3     if cond is None:
4         cond = fields
5
6     generate_create(table, fields, returning, prefix=prefix)
7     generate_delete(table, cond, prefix=prefix)
8     generate_get(table, prefix=prefix)
9     generate_set(table, fields, prefix=prefix)

```

0.8.4 Implementazione API

Una volta completata la libreria di supporto, è possibile implementare la logica dell'applicazione.

Di seguito, tutte le chiamate di cui si possono generare automaticamente gli endpoint.

Listing 15: Endpoint generati automaticamente

```

1 generate_crsd("categoria_attrezzatura", ["nome_categoria"])
2 generate_crsd("tipo_attivita", ["nome_tipo_attivita"])
3 generate_crsd("comune", ["nome_comune"])
4 generate_crsd("abilitazione", ["nome_abilitazione"])
5 generate_crsd("ha_abilitazione", ["nome_abilitazione", "numero_tessera"])
6 generate_crsd("abilitazioni_richieste", ["codice_attivita", "nome_abilitazione"])
7 generate_crsd("attivita_in_comune", ["codice_attivita", "nome_comune"])
8 generate_crsd("usa_consumabile", ["codice_attivita", "nome_attrezzatura", "quantita_utilizzata"],
9                 ["codice_attivita", "nome_attrezzatura"])
10 generate_crsd("usa_non_consumabile", ["codice_attivita", "codice_inventario", "rientrato",
11                  "stato_attuale_attrezzatura"], ["codice_attivita", "codice_inventario"])
12 generate_crsd("partecipazione", ["id_partecipante", "codice_attivita", "squadra", "timestamp_ingresso",
13                  "timestamp_uscita"], ["codice_attivita", "id_partecipante"])
14 generate_crsd("persona_esterna", ["nome", "cognome", "telefono"], ["id_partecipante"], returning="idPartecipante")
15 generate_crsd("disponibilita", ["codice_attivita", "numero_tessera",
16                  "disponibilita_positiva"], ["codice_attivita", "numero_tessera"])
17
18

```

```

19 generate_delete("attivita", ["codice_attivita"])
20 generate_delete("attivita_completata", ["codice_attivita"])
21 generate_delete("persona", ["numero_tessera"])
22 generate_delete("attrezzatura", ["id_attrezzatura"])
23
24
25 generate_get("persona")
26 generate_get("attrezzatura_consumabile", query="SELECT * from
    attrezzatura natural join attrezzatura_consumabile")
27 generate_get("attrezzatura_non_consumabile", query="SELECT * from
    attrezzatura natural join attrezzatura_non_consumabile")

```

Per concludere, sono presenti tutti gli endpoint complessi, non generabili automaticamente. Si noti come vengano comunque sfruttate le funzioni di libreria.

Listing 16: Esempio di endpoint non generabile

```

1 @app.route('/api/get_persona_dettagli', methods=["GET"])
2 def get_persona_dettagli():
3     #parametri in ingresso
4     numero_tessera = request.args.get("numero_tessera")
5     if numero_tessera is None:
6         return defaultToJson({"errore": "numero_tessera null"}), 400
7
8     def f(cursor): #codice che utilizza il database
9         cursor.execute(
10             'SELECT * from persona',
11             'where numero_tessera = %s', (numero_tessera, ))
12
13     res = cursor.fetchone()
14
15     if res is None:
16         return None
17
18     cursor.execute("select nome_abilitazione from
19                     ha_abilitazione where numero_tessera = %s",
20                     (numero_tessera, ))
21     res["ha_abilitazione"] = cursor.fetchall()
22
23     cursor.execute("select * from disponibilita "
24                   "natural join attivita "
25                   "where numero_tessera = %s",
26                   (numero_tessera,))
27     res["disponibilita"] = cursor.fetchall()
28
29     cursor.execute("select * from persona "
30

```

```

27                     "natural join partecipazione "
28                     "natural join attivita_completata natural
29                         join attivita "
30                     "where numero_tessera = %s" , (
31                         numero_tessera , ) )
30         res [ "partecipazione" ] = cursor . fetchall ( )
31
32
33     return res
34
35     return do ( f )

```

0.8.5 Autenticazione

L'autenticazione dell'utente avviene tramite credenziali [?]. Per poter essere riconosciuto dal sistema, l'utente deve fornire una coppia *username*, *password* valida.

Per avere un discreto livello di sicurezza, è bene richiedere l'utilizzo di password prive di significato, con lunghezza congrua (almeno 8-10 caratteri) e non utilizzate altrove.

Ad ogni profilo utente è associato un livello di autorizzazione. I livelli di autorizzazione sono

- *login*: l'utente ha un profilo. Default per la visualizzazione dei dati;
- *crea_evento*: l'utente può creare, modificare ed eliminare eventi. Utilizzato da coloro che si occupano dell'inserimento dati;
- *cms*: può effettuare attività di content management (ad esempio modificare l'elenco dei comuni o creare una nuova persona).

L'accesso ad endpoint API protetti avviene tramite *access token*. L'utente può ottenere un nuovo access token tramite una richiesta di *login*, alla quale fornisce *username* e *password*. La richiesta di *logout* elimina permanentemente l'access token.

Password, access token e autorizzazioni sono conservati in due tabelle dedicate nel database: *auth* per password e autorizzazioni, *token* per gli access token.

Listing 17: tabelle auth e token

```

1  create table auth(
2      email text primary key,
3      password text not null,
4      numero_tessera text,
5      crea_evento boolean not null,
6      cms boolean not null,
7      foreign key (numero_tessera) references persona(numero_tessera)
        ) on update cascade on delete cascade
8 );
9
10 create table token(
11     token text primary key,
12     email text not null,
13     foreign key (email) references auth(email) on update cascade
        on delete cascade
14 );
15
16 — default admin
17 insert into auth(email, password, numero_tessera, crea_evento, cms)
    values ('admin', 'admin', null, True, True);

```

L'endpoint *do_login* si occupa di effettuare il login.

Email e password sono ricercati nella tabella *auth*. Se non ci sono corrispondenze, la chiamata ritorna errore *401 Unauthorized*. Se invece viene trovata una corrispondenza (che per forza è univoca, essendo *email* chiave univoca), si procede alla generazione del token.

Il token è composto dal numero tessera concatenato ad una stringa casuale di lunghezza 32. Il numero tessera prefisso permette di identificare il proprietario del token. La stringa casuale viene scelta in modo da avere solo caratteri inseribili in un URL, per evitare di doverla ricodificare nella risposta.

Una volta generato il token, lo si inserisce nella tabella *token* associandolo all'email dell'utente (chiave primaria di *auth*).

Il token viene incluso nel corpo della risposta, in modo da poter essere utilizzato dal client.

Listing 18: POST do_login

```

1 @app.route("/api/do_login", methods=["POST"])
2 def doLogin():

```

```

3     data = getDataSettingNull()
4
5     def f(cursor):
6         cursor.execute("select * from auth where email=%s and
7                         password=%s", (data[ "email" ], data[ "password" ]))
8         row = cursor.fetchone()
9
10        if row is None:
11            return { "message": "not found", "token": None}, 401
12
13        row[ "token" ] = str(row[ "numero_tessera" ]) + secrets.
14            token_urlsafe(32)
15
16        cursor_insert(cursor, row, "token", [ "token", "email" ])
17
18        return { "message": "ok", "token": row[ "token" ]}
19
20    return do(f, auth=None)

```

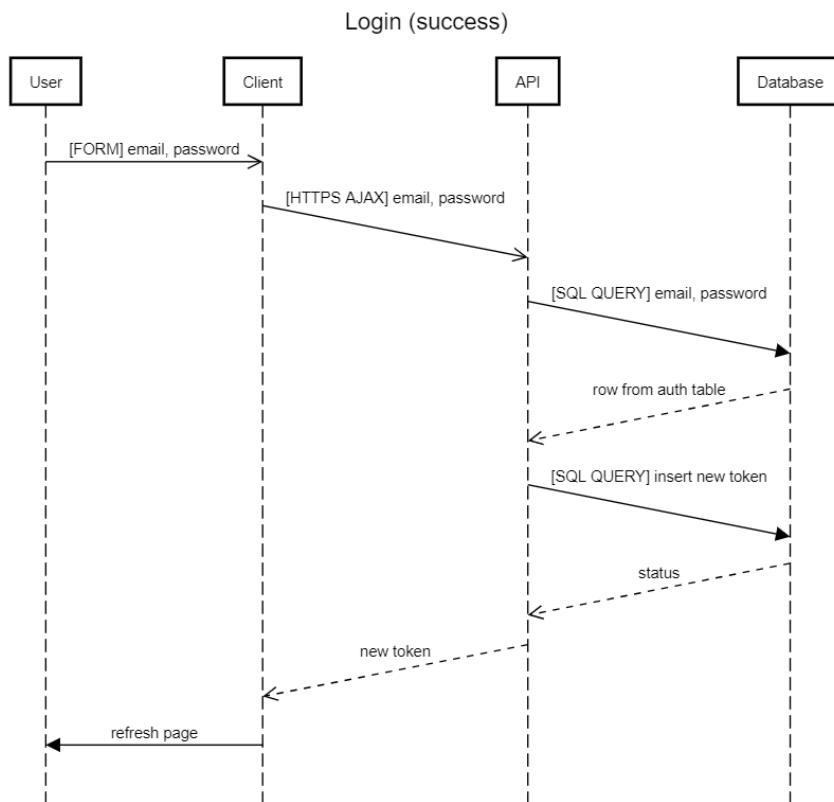


Figura 25: Creazione di un nuovo access token (successo)

Per associare la propria identità ad una richiesta, il client scrive il valore del proprio token nel campo *Authorization* della richiesta HTTPs.

La funzione ausiliaria *getLogin* si occupa di ottenere i dati e le autorizzazioni dell’utente che ha effettuato una richiesta. Per far ciò, legge il token dall’header *Authorization* e lo ricerca nel database.

Listing 19: funzione ausiliaria *getLogin*

```

1 def getLogin(cursor):
2     token = request.headers.get("Authorization")
3
4     cursor.execute("select * from token natural join auth where
5         token=%s", (token, ))
6     row = cursor.fetchone()
7
8     if row is not None:
9         del row["password"] #nasconde la password per maggiore
10            sicurezza
11
12     return row

```

La funzione di supporto *do* è stata modificata per includere il controllo di autorizzazione.

Essa, prima di effettuare la query, controlla che il livello di autorizzazione dell’utente sia adeguato. Se il parametro *auth* è una stringa, identifica il nome del livello di autorizzazione. Se esso è *None*, il controllo delle autorizzazioni è disabilitato. Se *auth* è una funzione, viene interpretato come una funzione booleana che, date le autorizzazioni e i dati dell’utente, restituisce *True* se l’utente è autorizzato.

Nel caso in cui l’utente non disponga delle autorizzazioni necessarie, viene inviata una risposta contenente un codice d’errore adeguato (401 se il token non è valido, 403 se non dispone di sufficienti autorizzazioni).

Listing 20: Funzione ausiliaria *do* modificata

```

1 def do(query, applyRes=defaultToJson, auth="cms"):
2     cnx = cnxpool.getconn()
3     with cnx.cursor(cursor_factory=psycopg2.extras.RealDictCursor)
4         as cursor:
5             if auth is not None:
6                 permissions = getLogin(cursor)
7                 if permissions is not None: permissions["login"] =
8                     True

```

```

7      if isinstance(auth, str):
8          if permissions is None:
9              return applyRes({"message": "E' necessario
10                 effettuare il login"}, 401)
11         if not permissions[auth]:
12             return applyRes({"message": "E' necessaria un'
13                 autorizzazione di tipo " + auth}, 403)
14     else:
15         auth(cursor, permissions)
16
17     res = query(cursor)
18
19     res = applyRes(res)
20     cnx.commit()
21     cnxpool.putconn(cnx)
22
23     return res

```

Le funzioni di supporto per la generazione automatica di endpoint sono state modificate in modo da includere l'autorizzazione richiesta.

Listing 21: Esempio di funzione generatrice di endpoint modificata e relativa chiamata

```

1 def generate_create(table, fields, returning, auth="cms", prefix="/api"):
2     def f_create():
3         ...
4
5     def f(cursor):
6         ...
7
8     return do(f, auth=auth)
9
10 ...
11
12
13 def generate_crsd(table, fields, cond=None, returning=None, auth="cms",
14     auth_get="login", prefix="/api"):
15     if auth_get == "same":
16         auth_get = auth
17
18     if cond is None:
19         cond = fields
20
21     generate_create(table, fields, returning, auth=auth, prefix=
22                     prefix)

```

```

21     generate_delete(table , cond , auth=auth , prefix=prefix)
22     generate_get(table , auth=auth_get , prefix=prefix)
23     generate_set(table , fields , auth=auth , prefix=prefix)
24     generate_update(table , fields , cond , returning , auth=auth ,
25                       prefix=prefix)
26
27 generate_crsd( " categoria_attrezzatura " , [ " nome_categoria " ] , auth="
28   cms" )

```

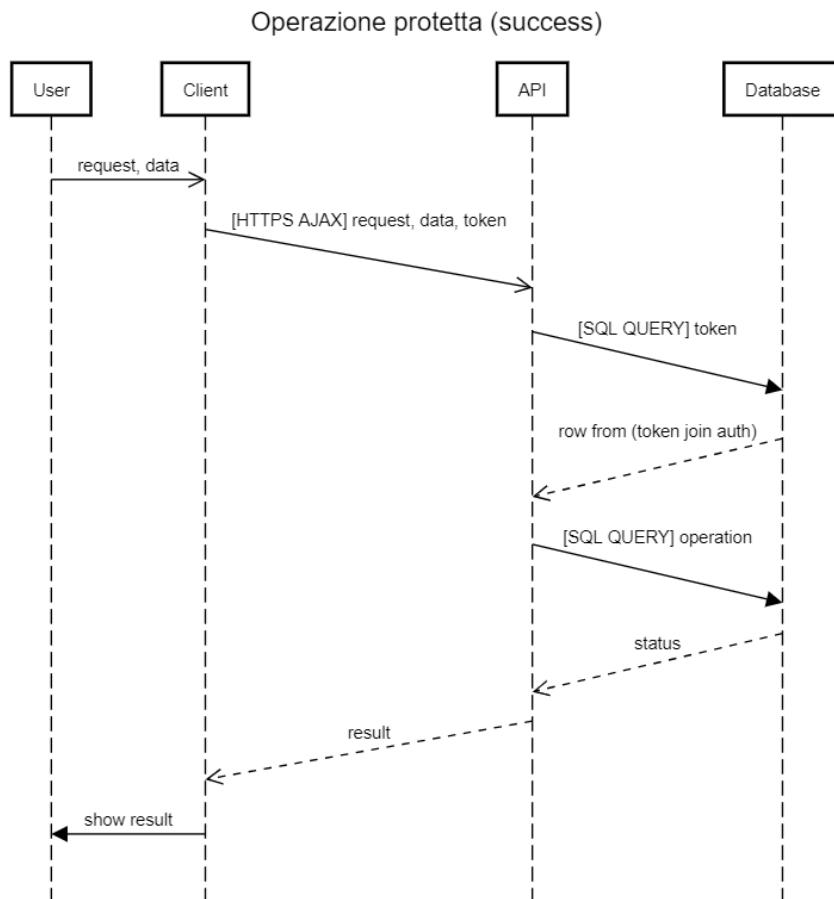


Figura 26: Esecuzione di una richiesta protetta (successo)

Il logout è effettuabile mediante l'endpoint *do_logout*. L'unico parametro richiesto è il token da eliminare.

Listing 22: POST do_logout

```
1 @app.route( "/api/do_logout" , methods=[ "POST" ])
```

```

2 def doLogout():
3     token = request.headers.get("Authorization")
4     if token is None:
5         return {"res": "token null"}, 400
6
7     def f(cursor):
8         cursor.execute("delete from token where token = %s
9                         returning token", (token, ))
10
11    return {"message": "ok", "data": cursor.fetchone()}
12
13 return do(f, auth=None)

```

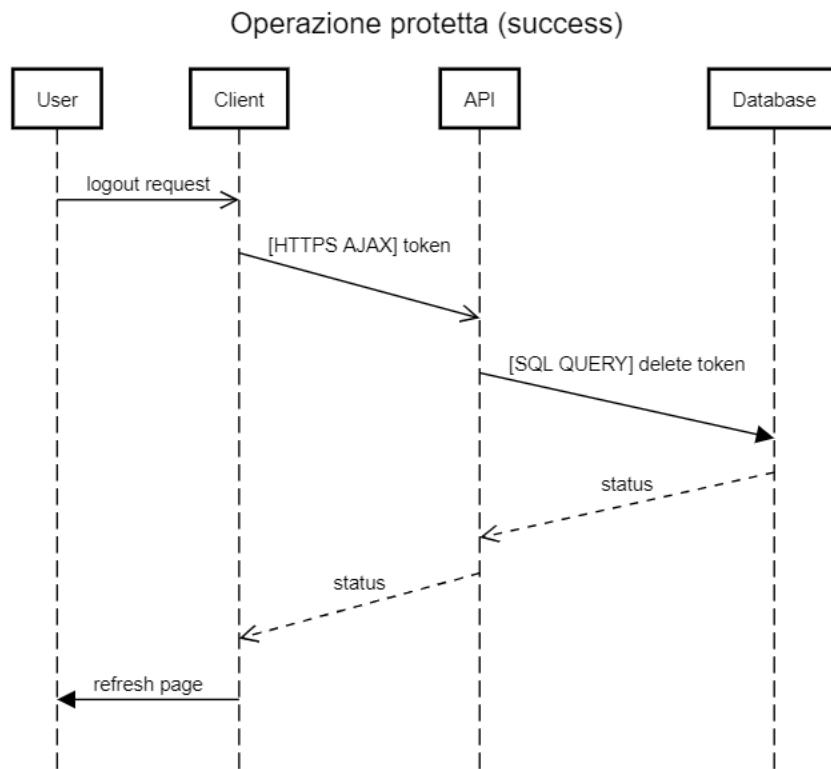


Figura 27: Logout ed eliminazione token (successo)

Il recupero della password è possibile solo tramite email.

Tramite richiesta all'endpoint *richiedi_recupera_pw*, il sistema invia un'email all'indirizzo specificato contenente un link alla pagina *recupera_pw*.

All'interno del link viene generato un token univoco, associato all'utente. Esso è a tutti gli effetti un access token legato alla email specificata, con cui il proprietario può modificare la password o effettuare altre operazioni.

Listing 23: endpoint richiedi_recupera_pw

```

1 @app.route( "/api/richiedi_recupera_pw" , methods=[ "POST" ] )
2 def get_recuperaPw():
3     data = getDataSettingNull()
4
5     def f(cursor):
6         d = dict()
7         d[ "token" ] = secrets.token_urlsafe(20)
8         d[ "email" ] = data[ "email" ]
9
10        cursor_insert(cursor , d , "token" , [ "token" , "email" ])
11
12        ...
13
14        me, you = "pc.cms.sanvito@gmail.com" , d[ "email" ]
15
16        msg = MIMEText( 'alternative' )
17        msg[ 'Subject' ] = "Link"
18        msg[ 'From' ] = me
19        msg[ 'To' ] = you
20
21        link = "http://localhost:8080/render/recupera_pw?token=" +
22               d[ "token" ]
23
24        text = "Segui il link per recuperare la password\n" + link
25        html = """\
26            <html>
27                <head></head>
28                <body>
29                    <p>Recupero password<br>
30                        Segui il link per recuperare la password<br>
31                        <a href=""" + link + """>link</a>
32                    </p>
33                </body>
34            </html>
35
36            ...
37
38        server = smtplib.SMTP(smtp_server , port)
39        server.starttls(context=context) # Secure the connection
40        server.login(sender_email , password)
41        server.sendmail(me, you, msg.as_string())
42        server.quit()

```

```

43         return { "message": "ok" }
44
45     return do(f, auth=None)

```

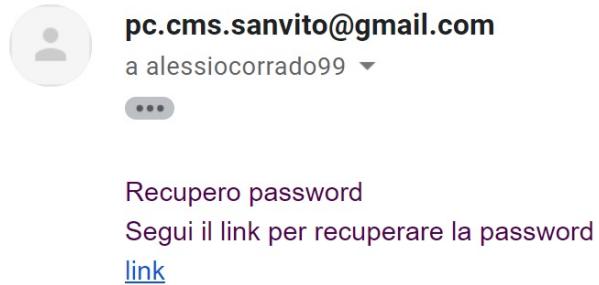


Figura 28: Email di recupero password

L'endpoint *recupera_pw* si occupa di impostare la nuova password per l'utente, se la richiesta è valida.

Il recupero password elimina tutti i token associati all'utente. In questo modo, il recupero password permette di eliminare eventuali token rubati ed evitare che chi conosceva la vecchia password possa continuare ad operare.

Listing 24: POST *recupera_pw*

```

1 @app.route("/api/recupera_pw", methods=["POST"])
2 def do_recuperaPw():
3     token = request.headers.get("Authorization")
4     data = getDataSettingNull()
5
6     def f(cursor):
7         cursor.execute("select * from token natural join auth
8             where email=%s and token=%s", (data["email"], token))
9         row = cursor.fetchone()
10
11         if row is None:
12             return {"message": "not found"}, 401
13
14         data["email_old"] = data["email"]
15         cursor_update(cursor, data, "auth", ["password"], ["email"])
16         cursor_delete(cursor, data, "token", ["email"])
17
18         return {"message": "ok"}

```

```
18
19     return do(f, auth=None)
```

0.8.6 Testing

Il testing è immediato da eseguire: Flask mette a disposizione un suo server di test, richiamabile includendo all'interno dello script la funzione *run*. La funzione di debug permette di leggere agevolmente i log e risalire ai punti in cui vengono lanciate delle eccezioni.

Per poter interfacciarsi con il server è consigliabile utilizzare un *client REST*. Quello scelto dallo sviluppatore è *Insomnia* [?].

Nel caso specifico, il testing è stato eseguito provando a richiamare ogni endpoint e passando diverse combinazioni di dati, sia validi che non, verificando che i risultati fossero congrui alle specifiche.

```
Run: main.py
C:\Users\aless\miniconda3\envs\protezioneCivile_web\python.exe C:/Users/aless/Dropbox/protezioneCivile_web/web/main.py
redis= None None
↓
{'user': 'postgres', 'password': 'postgres', 'database': 'postgres', 'host': '172.29.9.255'}
    * Serving Flask app "main" (lazy loading)
    * Environment: production
      WARNING: This is a development server. Do not use it in a production deployment.
      Use a production WSGI server instead.
    * Debug mode: on
    * Restarting with stat
redis= None None
{'user': 'postgres', 'password': 'postgres', 'database': 'postgres', 'host': '172.29.9.255'}
    * Debugger is active!
    * Debugger PIN: 458-744-411
    * Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
127.0.0.1 - - [08/Mar/2021 13:35:40] "GET /api/get_persona_dettagli?numero_tessera=4485713447012018 HTTP/1.1" 200 -
```

Figura 29: Console di debug per Flask

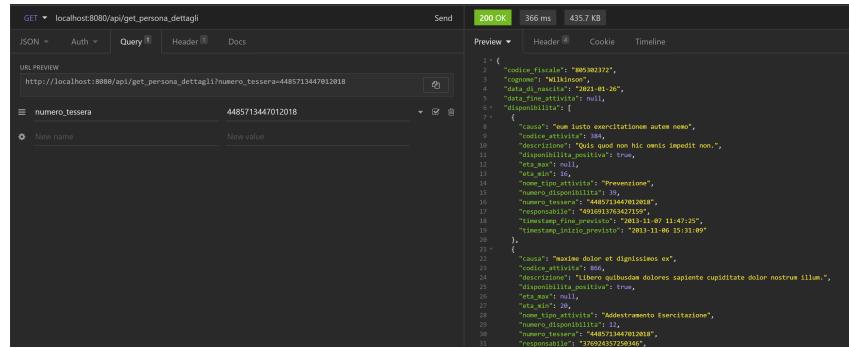


Figura 30: Utilizzo del client REST Insomnia

0.8.7 Deploy locale

Il deploy è stato nuovamente effettuato creando un’istanza Docker. Come server di produzione è stato scelto di usare *gunicorn*, uno dei più famosi per Flask.

Per creare il container, è necessario partire da un’immagine contenente python3+gunicorn, installare le librerie necessarie e copiare i sorgenti.

Listing 25: Dockerfile

```
1 FROM tiangolo/meinheld-unicorn:python3.7
2 COPY requirements.txt /
3 RUN pip install -r /requirements.txt
4 COPY . /app
```

Listing 26: File di configurazione per Docker compose

```
1 web:
2   container_name: protezionecivile_web
3   image: protezionecivile_web
4   environment:
5     SQL_USERNAME: postgres
6     SQL_PASSWORD: postgres
7     SQL_DATABASE_NAME: postgres
8     SQL_HOST: 172.29.9.255
9   volumes:
10    - ./web:/app
11   ports:
12    - "<porta esterna >:80"
```

0.9 Suddivisione dei compiti tra app e website

Applicazione e sito web sono complementari. Per questo sorge spontanea una domanda: per ogni funzionalità, dove metterla?

Dopo aver dialogato con il committente è risultato chiaro che la quasi totalità dei membri preferisce accedere ai contenuti tramite smartphone. Per questo motivo tutte le funzionalità "operative" vanno sicuramente incluse all'interno dell'app.

Per quanto riguarda invece la parte di *content management*, lo schermo di maggiori dimensioni, unito alla possibilità di consultare in parallelo fogli excel e altri tipi di documento, suggeriscono che l'utilizzo da PC sia preferibile. Queste operazioni inoltre vengono svolte di rado (al massimo un paio di volte l'anno) e solamente da pochi membri designati come responsabili.

Quello che rimane è la consultazione dei grafici interattivi e statistiche varie. Risulta quindi opportuno inserire queste funzionalità sia nell'applicazione, magari in formato ridotto, sia nell'interfaccia web, con la possibilità di una maggiore possibilità di personalizzazione.

0.10 App

Lo sviluppo dell'app è una delle parti più ostiche. Oltre alle difficoltà di programmazione, è importante anche andare incontro ai gusti degli utilizzatori, per fornire una buona estetica ed usabilità.

In questa sezione verrà affrontato l'intero processo di sviluppo dell'app, che quindi conterrà anche una parte meno "tecnica", ma altrettanto necessaria: il *design* [?].

0.10.1 Design

Inanzitutto, è stato fissato un incontro con il committente incentrato sul design dell'applicazione. Sposando la filosofia "user-centered", è importante includere il più possibile l'utilizzatore finale, in particolare per quanto riguarda il design di interfaccia.

Inizialmente si sono identificate le azioni che l'utente svolge durante il lavoro.

Lista dei task

- 1 Effettuare il login/logout
 - 2 Fornire la disponibilità per un'attività
 - 3 Vedere i dettagli di un'attività
 - 4 Vedere la lista delle attività future
 - 5 Modificare una delle proprie disponibilità
 - 6 ...
 - 7
 - 8 SOLO PER I COORDINATORI
 - 9 Creare un'attività
 - 10 Gestire le presenze di un'attività in svolgimento
 - 11 Terminare e registrare un'attività in svolgimento
 - 12 Gestire i materiali di un'attività
 - 13 ...
-

Mediante la tecnica di *task analysis*, le azioni sono state scomposte in *passaggi* semplici.

Task analysis

- 1 EFFETTUARE IL LOGIN
- 2 Se l'utente ha salvato le credenziali:
 - 3 Effettua il login automaticamente e vai alla schermata principale
- 4 Altrimenti:

5 Inserisci nome utente
6 Inserisci password
7 Effettua login
8 Se login corretto:
9 Vai alla schermata principale
10 Altrimenti:
11 Riprova
12
13
14 FORNIRE LA DISPONIBILITA' PER UN'ATTIVITA'
15 Visualizza la lista di attivita'
16 Scegli l'attivita' a cui dare la disponibilita'
17 Inserisci l'orario della disponibilita'
18 Conferma l'inserimento della disponibilita'
19
20
21 INSERIRE UNA PRESENZA PER UN'ATTIVITA' IN SVOLGIMENTO
22 Visualizza le attivita' in svolgimento
23 Scegli l'attivita' a cui aggiungere la presenza
24 Scegli la persona di cui inserire la presenza
25 Inserisci l'orario di arrivo
26 Conferma l'inserimento della presenza
27
28
29 CREARE UN'ATTIVITA'
30 Inserisci il nome dell'attivita'
31 Scegli il tipo dell'attivita'
32 Inserisci la causa dell'attivita'
33 Inserisci la descrizione dell'attivita'
34 Inserisci la data di svolgimento dell'attivita'
35 Inserisci l'ora di inizio prevista dell'attivita'
36 Inserisci l'ora di fine prevista dell'attivita'
37 Se ci sono vincoli di eta':
38 Inserisci l'eta' minima di partecipazione
39 Inserisci l'eta' massima di partecipazione
40 Scegli i comuni in cui si svolgera' l'attivita'
41 Scegli le abilitazioni richieste per lo svolgimento dell'attivita'
42 Scegli il responsabile dell'attivita'
43 Conferma l'inserimento della nuova attivita'
44
45 ...

Si noti la scelta dei verbi nella task analysis:

- *Scegli*: scelta da un insieme di elementi possibili;
- *Inserisci*: inserimento da parte dell'utente;

- *Effettua*: compimento di un'azione che non prevede scelte o inserimenti;
- *Vai*: cambio di contesto;
- *Conferma*: conferma esplicita al termine di una procedura.

Le azioni sono state raggruppate in *concetti*, organizzati in modo gerarchico. Il grafo dei concetti è la base per creare la struttura della *navigazione*.

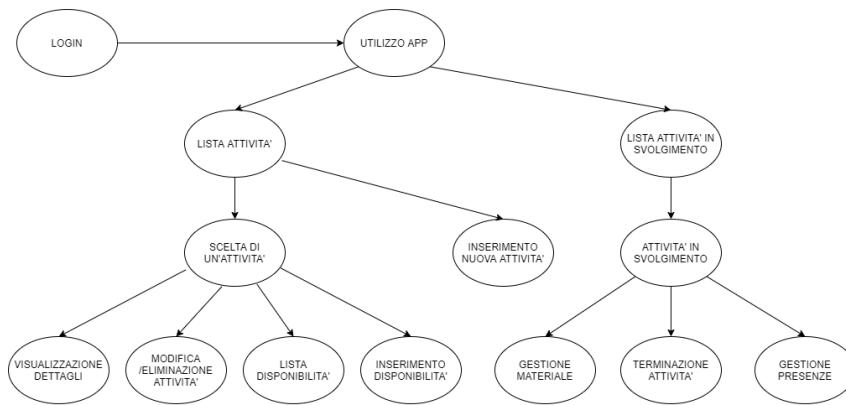


Figura 31: Grafo dei concetti

Come interfaccia di navigazione principale. è stato scelto di usare una sidebar. Il suo contenuto è il seguente:

- Nome e tipologia profilo
- Home
- Storico
- Statistiche
- Impostazioni
- Calendario
- Logout

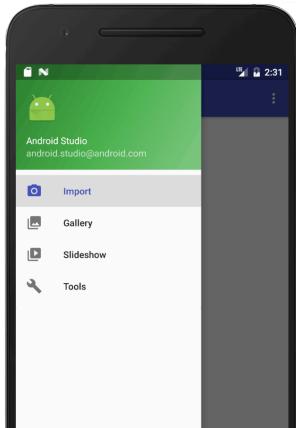


Figura 32: Esempio di sidebar

Sono stati quindi creati i wireframe: rappresentazioni grafiche delle schermate dell'app, collegate tramite frecce. All'interno dei wireframe ci sono anche brevi descrizioni dei componenti grafici.

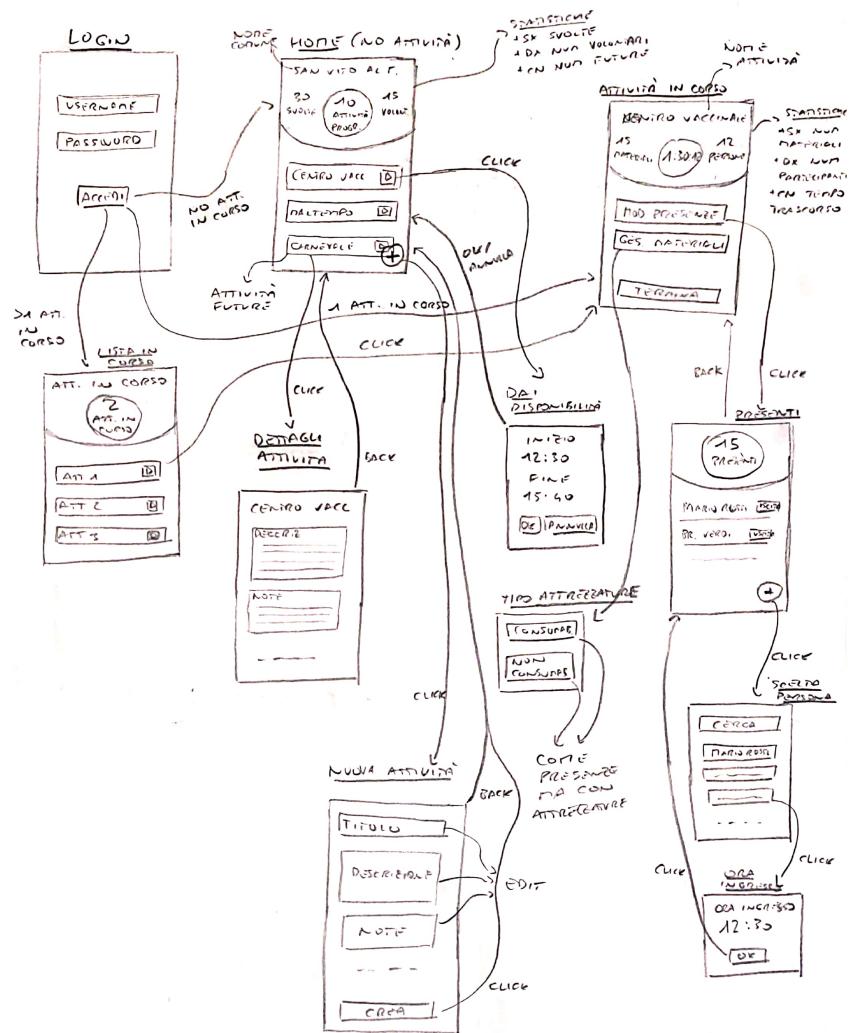


Figura 33: Wireframe dell'app

Approfondimento: User-centered design

Lo *user-centered design* [?] è una particolare filosofia di design che incentra la sua attenzione sugli utenti.

Un punto chiave è il *design partecipato*: gli utilizzatori vengono chiamati a partecipare al processo di design e sviluppo, in modo da dare il loro massimo contributo. Per produrre un’interfaccia che piaccia ad una persona non c’è modo migliore che chiedere a lei stessa!

Nella prima fase si identificano gli obiettivi: quali sono le competenze degli utilizzatori, qual è l'ambiente di utilizzo, quali sono le richieste specifiche. È importante esaminare queste informazioni, perché per utenti e contesti diversi possono essere adatte interfacce diverse. Ad esempio, una calcolatrice usata da un ingegnere è ben diversa da quella usata da un bambino.

La *task analysis* è il processo che analizza attentamente lo svolgimento dei task, per scomporli in azioni fondamentali. Si possono usare diversi livelli di granularità, in base a quale sia lo scopo dell'analisi.

L'utilizzo di *mock-up* permette di mostrare all'utente, fin dalla fase di progettazione, quello che sarà l'aspetto del prodotto in modo da poterlo giudicare.

Gli *scenari* permettono di simulare lo svolgimento delle azioni e l'interazione con il sistema. La sequenza di azioni nello scenario può essere usata per determinare il grafo di transizione tra le schermate.

L'utilizzo di *guideline* è utile per produrre un'interfaccia consistente con quelle presenti nello stesso contesto. Esistono diversi livelli di guideline: alcuni esempi sono quelle per le app Android, quelle per le app iOS, quelle per applicazioni desktop Windows. Inoltre permettono di riutilizzare componenti standard, in modo da non dover "reinventare la ruota".

Le valutazioni basate sull'*osservazione degli utenti* permettono agli sviluppatori di avere un ottimo *feedback* sull'interfaccia che hanno sviluppato, in modo da poterla migliorare.

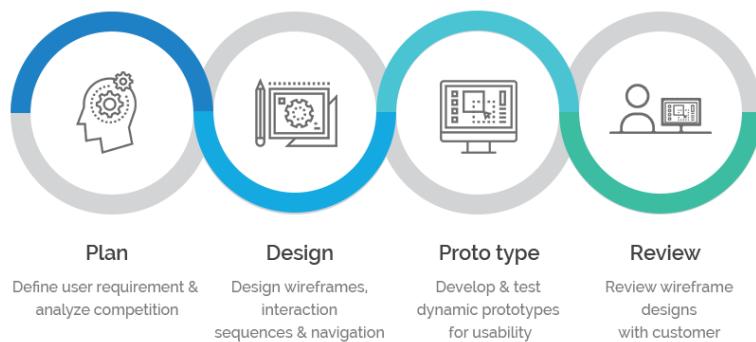


Figura 34: User-centered design

0.10.2 Progettazione

Come piattaforma di sviluppo è stato scelto di utilizzare il framework *React Native* [?]. Esso permette di codificare una volta sola per ottenere contemporaneamente una versione adatta ad *Android* e una ad *iOS*.

L'unità principale di sviluppo con React Native è il *componente*. In genere, ad ogni schermata dell'app corrisponde un componente diverso. In più, la singola schermata può essere suddivisa gerarchicamente in componenti: una lista di attività, un elemento di questa lista, un'immagine, ecc ...

Per ogni schermata, è stata definita la gerarchia dei suoi componenti. Di seguito un esempio.

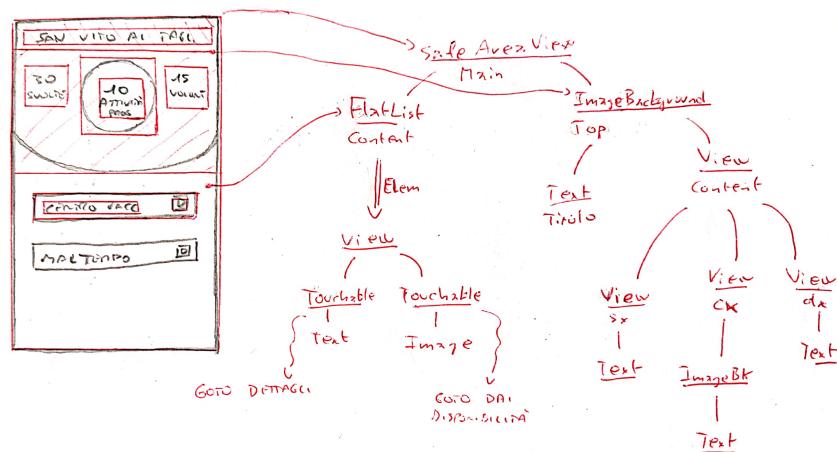


Figura 35: Gerarchia schermata "Home_no_attivita"

I componenti si compongono utilizzando l'inclusione. Il componente padre configura i componenti figli tramite il passaggio di dati mediante *props*. I figli interagiscono verso il padre mediante l'esecuzione di *callback*.

Secondo il principio di *single source of truth*, ogni dato va mantenuto in un solo luogo, che diventa la "*sola e unica fonte di verità*". Ciò permette di evitare duplicazioni dei dati, con conseguenti perdite di consistenza.

Ogni dato va salvato nel *least common ancestor* dei componenti che lo usano. L'hook *val, setter = useState(initVal)*, richiamato dal componente che

memorizza il dato, permette di ottenerne getter e setter sotto forma di callback, i quali sono passati ai figli che ne hanno bisogno.

Per ogni componente, sono stati elencati i dati da esso utilizzati. A seguire un esempio.

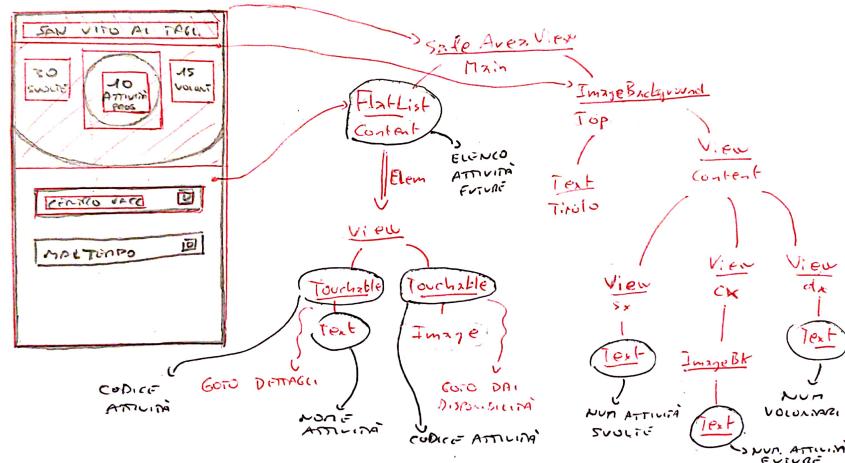


Figura 36: Dati utilizzati dai componenti di "Home_no_attivita"

È stato creato un componente che gestisce la navigazione principale, il quale contiene tutti i componenti della sidebar individuati nella fase di design.

Per gestire la navigazione delle singole sezioni, si è deciso di utilizzare degli *StackNavigator*. Essi permettono di utilizzare il pulsante "indietro" per navigare all'indietro, pratica comune sia in Android che iOS.



Figura 37: Struttura navigazione principale

0.10.3 Implementazione

Una volta terminata la progettazione di una schermata, è possibile passare alla sua implementazione.

Il primo passaggio consiste nel creare un nuovo file, il quale conterrà il codice relativo al nuovo componente. Vengono inanzitutto scritti gli import e la struttura di base del componente.

Come prima cosa si implementa l'interfaccia utente, seguendo la decomposizione in moduli definita nella fase di progettazione.

In questa fase i dati vengono inseriti direttamente nel sorgente, in quanto servono solamente per avere una prima resa grafica.

I componenti sono grezzi, privi di dettagli. Vengono scelti colori di sfondo contrastanti per avere una buona identificazione di dove si trova ciascun componente.



Figura 38: Prima versione della schermata - i dati sono finti

Una volta inclusi e testati tutti i componenti, si passa alla visualizzazione di dati "veri".

Per caricare i dati dall'API si usa un *test driver*: un modulo di codice che simula la presenza di un chiamante e che carica i dati in modo "statico" (ad esempio, per caricare la lista delle attività viene usato un token di accesso generato esternamente all'app, ma solo inserito nel codice, in quanto dovrebbe essere generato dalla schermata di login).



Figura 39: I dati sono caricati dall'API

A questo punto si implementano i gestori degli eventi, come ad esempio il click su un elemento della lista. Questi gestori inizialmente mostrano solo un messaggio di avviso; in un secondo momento inviano i dati al server (se necessario) e usano l'interfaccia di navigazione per passare alla prossima schermata.

Si rifiniscono quindi i colori e dettagli grafici per ottenere la versione finita del componente.



Figura 40: Scelta colori e dettagli grafici

Nell'ultima fase il test driver viene rimosso e la schermata viene integrata assieme al resto dell'app.

Ad ogni integrazione, una approfondita fase di test mira a rimuovere tutti i difetti. Prima vengono eseguite le correzioni, più è probabile che esse siano circoscritte (non si diffondono ad altri componenti).

0.10.4 Testing

Il testing dell'app è stato effettuato tramite la piattaforma Expo [?]. Essa permette di mettere in comunicazione il dispositivo mobile con il server di sviluppo, caricando automaticamente tutte le modifiche *on-the-fly* e gestendo il *logging/debugging*.

La connessione può avvenire sia su rete locale, sia da remoto (tramite tunnel pubblico creato da Expo verso il dispositivo di sviluppo). In questo modo è possibile eseguire ed esaminare il comportamento dell'app direttamente sul dispositivo del committente o di un tester specializzato.

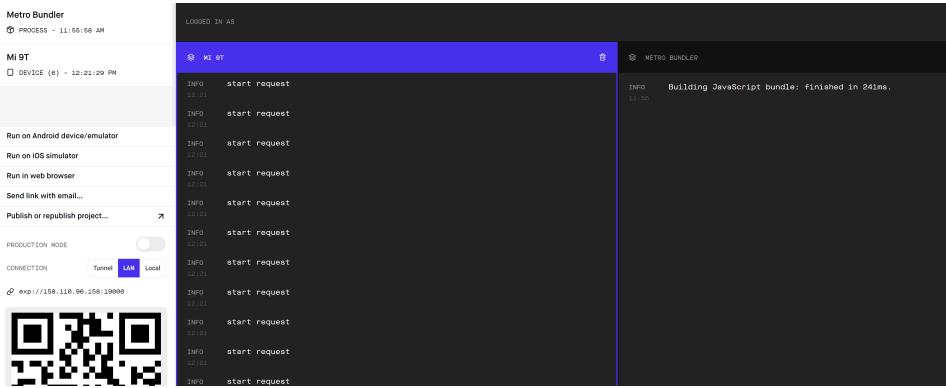


Figura 41: Testing - Dispositivo di sviluppo

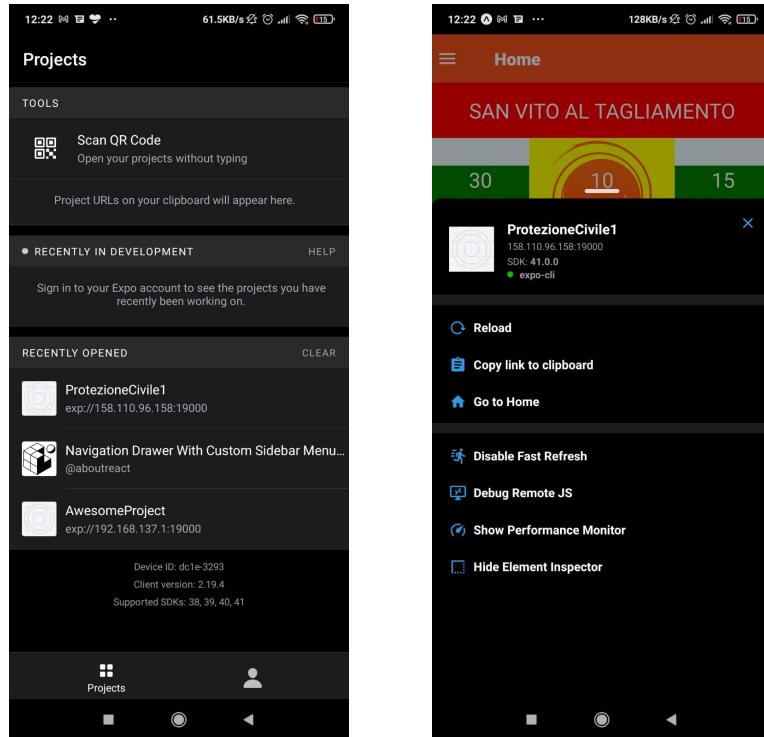


Figura 42: Testing - dispositivo mobile

Sempre utilizzando Expo, è possibile pubblicare i propri progetti sul web. Basta compilare un breve form contenente nome dell'app e dati dello sviluppatore affinché Expo metta a disposizione automaticamente uno spazio in cloud.

0.11 Website

Questo capitolo si occuperà del processo di sviluppo del sito web.

Sebbene esso sia utilizzato meno frequentemente dell'app, e per questo necessiti di minor cura dei dettagli, è comunque importante rispettare quanto stabilito nella *politica di qualità*. Infatti, pur avendo un'interfaccia meno personalizzata, deve comunque essere utilizzabile con estrema semplicità (personale non specializzato) ed efficienza.

0.11.1 Struttura

Il website è stato ulteriormente diviso in due parti: *frontend* e *backend* [?].

Il *backend* viene eseguito da un server web e si occupa di inviare al client le pagine da esso richieste.

Il *frontend* viene eseguito all'interno del browser. Esso è composto da un sorgente di pagina (HTML), da un documento di stile (CSS) e da un insieme di script (JavaScript).

Nel caso di questo progetto, il backend genera soltanto la struttura della pagina, mentre i dati vengono richiesti alle API dal frontend (mediante *Ajax* [?]). Questo per alleggerire il carico del server, che si limita a servire pagine statiche, e sfruttare le API già scritte per l'utilizzo da parte dell'App.

La libreria utilizzata per gestire la grafica è *Bootstrap* [?]. Essa è una tra le più attualmente utilizzate e supportate dalla community.

La libreria *jQuery* [?] invece è ottima per implementare la parte dinamica della pagina, permettendo di effettuare richieste ajax e manipolare il *DOM*¹⁷.

0.11.2 Design

Il primo passaggio della fase di design del sito web è, come nel caso dell'app, l'elencazione dei task che esso deve essere in grado di svolgere.

¹⁷Il *Document Object Model* è un'interfaccia ad albero che, all'interno di un browser web, permette l'interazione con i contenuti della pagina.

I task individuati sono i seguenti:

1. Creazione/modificazione/eliminazione di *persone*¹⁸;
2. Creazione/modificazione/eliminazione di *abilitazioni*;
3. Creazione/modificazione/eliminazione di *comuni*;
4. Creazione/modificazione/eliminazione di *tipi attività*;
5. Creazione/modificazione/eliminazione di *categorie attrezzatura*;
6. Creazione/modificazione/eliminazione di *attrezzature consumabili*;
7. Creazione/modificazione/eliminazione di *attrezzature non consumabili*.

Inoltre, per il corretto funzionamento del sito, è necessario introdurre la funzione di *login/logout*.

Si è scelto di implementare la navigazione utilizzando una *responsive sidebar*. Per ogni task della lista è presente la voce corrispondente nella sidebar.

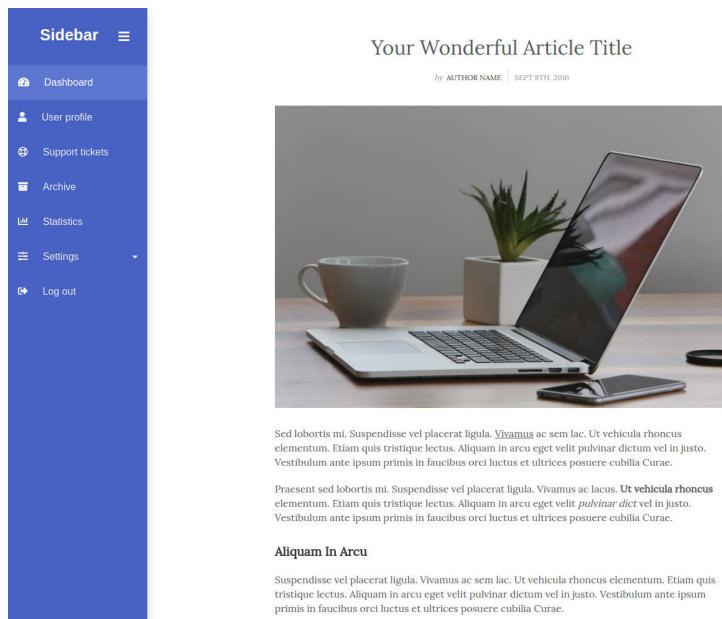


Figura 43: Esempio di navigazione con sidebar

¹⁸Si ricorda che con *persona* si intende un membro tesserato appartenente al gruppo locale di protezione civile.

Una volta definita la struttura di navigazione generale, sono stati creati i mockup relativi alle diverse pagine. Essi non devono essere riprodotti con una fedeltà assoluta, ma una loro verosimiglianza consente di valutare in modo migliore l'estetica del prodotto finale.

L'apposizione di appunti direttamente nel disegno di mockup permette di specificare meglio il comportamento o l'aspetto di componenti specifiche.

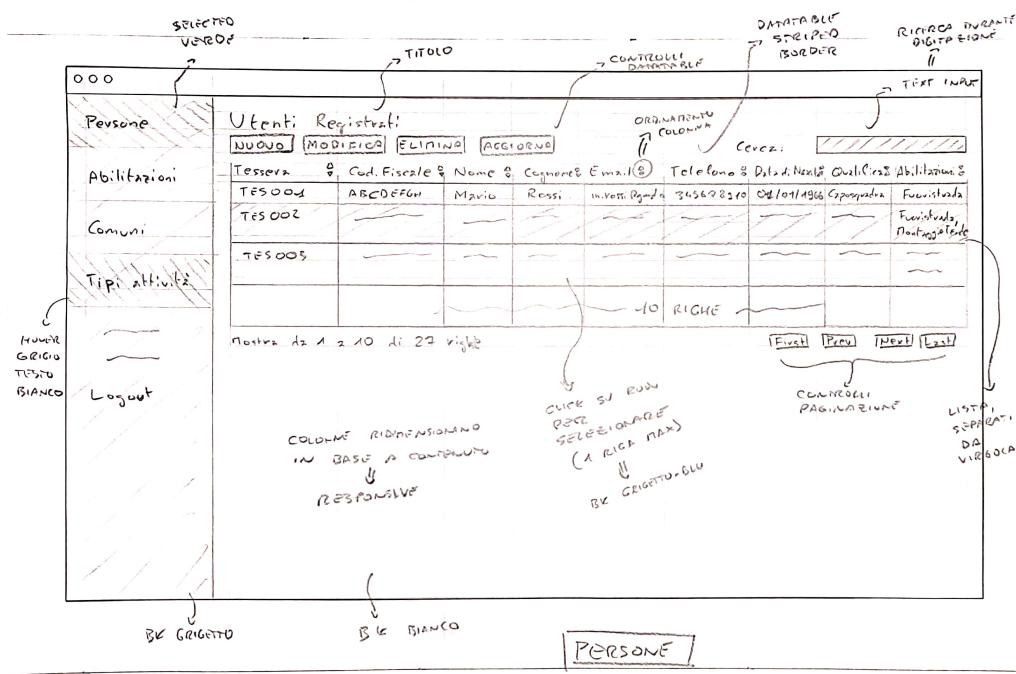


Figura 44: Esempio di mockup - pagina persone

I mockup vengono quindi utilizzati per effettuare una progettazione strutturale delle pagine: l'interfaccia viene divisa in componenti, ai componenti principali vengono assegnati dei tag o delle classi css ad-hoc.

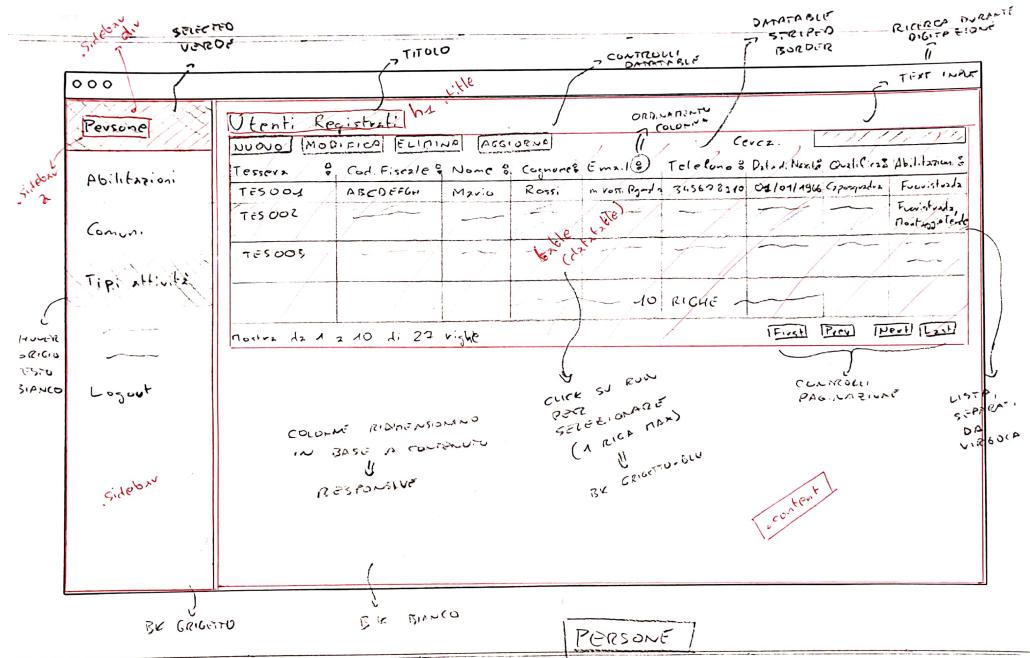


Figura 45: Esempio di mockup - struttura pagina persone

0.11.3 Implementazione

Per modularizzare il backend, è risultato conveniente sfruttare il *template engine* di Flask, *jinja 2* [?]. In questo modo è possibile modificare i contenuti in un solo file per cambiarli in tutte le pagine che lo includono.

Il template base è quello da cui derivano tutte le pagine. Esso è diviso in moduli: *title* per il titolo della pagina, *includes* per gli script e stylesheet esterni alla pagina, *content* per il contenuto principale della pagina, *scripts* per gli script interni.

Oltre alla struttura generale, il template base si occupa anche di richiedere ed effettuare il login dell'utente, se necessario, e di gestire il logout.

Listing 27: Template base.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
```

```
5      ...
6
7      <title>
8          {% block title%} Protezione Civile San Vito Al Tagliamento
9              {% endblock title %}
10
11     <link
12         rel="stylesheet"
13         href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
14             bootstrap.css"
15     />
16     ... altri link a stylesheet ...
17
18     {% block includes%} {% endblock includes %}
19
20     <style>
21         #recupera_pw_link {
22             float: left;
23         }
24         .sidebar {
25             margin: 0;
26             padding: 0;
27             width: 200px;
28             background-color: #f1f1f1;
29             position: fixed;
30             height: 100%;
31             overflow: auto;
32         }
33         /* Sidebar links */
34         .sidebar a {
35             display: block;
36             color: black;
37             ...
38         }
39
40         /* On screens that are less than 700px wide, make the
41            sidebar into a topbar */
42         @media screen and (max-width: 700px) {
43             .sidebar {
44                 width: 100%;
45                 height: auto;
46                 position: relative;
47             }
48
49             ... altre classi...
50     </style>
```

```
51     </head>
52 <body>
53     <div class="sidebar">
54         <a href="/render/persone">Persone</a>
55         <a href="/render/abilitazioni">Abilitazioni</a>
56         ... altri link...
57
58         <div class="" onclick="logout()">Logout</div>
59     </div>
60
61     <div class="content pt-5">{% block content %} {% endblock
62         content %}</div>
63
64     <div
65         class="modal fade"
66         id="modal_login"
67         tabindex="-1"
68         role="dialog"
69         aria-labelledby="exampleModalLabel"
70         aria-hidden="true"
71     >
72
73         <!-- dialog login -->
74         <div class="modal-dialog" role="document">
75             <div class="modal-content">
76                 <div class="modal-header">
77                     <h3 class="modal-title" id="exampleModalLabel">
78                         Effettua l'accesso
79                     </h3>
80
81                     ....
82             </div>
83
84             <script
85                 src="https://code.jquery.com/jquery-3.6.0.min.js"
86                 integrity="sha256-/xUj+3OJU5yExlq6GSYGSShk7tPXikynS7ogEvDej/
87                 m4="
88                 crossorigin="anonymous"
89             ></script>
90             ... altri script...
91
92             <script>
93
94                 $(document).ready(function () {
95                     //attiva link di pagina corrente
96                     $(".sidebar a").each(function () {
97                         if (window.location.href.includes(this.href)) {
98                             $(this).addClass("active");
99                         }
100                     });
101
102             </script>
```

```
98
99     modal_login = $("#modal_login");
100
101    token = Cookies.get("token");
102    if (token == null) {
103        showLogin();
104    }
105);
106
107 // mostra modal login
108 function showLogin() {
109     Cookies.remove("token");
110
111     modal_login.modal({
112         backdrop: false,
113         keyboard: false,
114     });
115 }
116
117 // oggetto contenente i dati inseriti nella login-form
118 function getLoginForm() {
119     return $("#login-form").serializeJSON();
120 }
121
122 function login_click() {
123     $.ajax({
124         url: "/api/do_login",
125         type: "POST",
126         dataType: "json",
127         contentType: "application/json",
128         data: JSON.stringify(getLoginForm()),
129     }).success(function (res) {
130         if (res.token == null) {
131             alert("credenziali errate");
132         } else {
133             token = res.token;
134             Cookies.set("token", token);
135             location.reload();
136         }
137     });
138 }
139
140 function logout() {
141     $.ajax({
142         url: "/api/do_logout",
143         ...
144         beforeSend: function (request) {
145             request.setRequestHeader("Authorization", token);
146         },
147     });
148 }
```

```

147         }) . success (function (res) {
148             Cookies.remove("token");
149             location.reload();
150         });
151     }
152
153     function recupera_pw_click() {
154         if (!confirm("Vuoi davvero recuperare la tua password?"))
155             return;
156
157         var data = getLoginForm();
158
159         if (data.email == "") {
160             alert("inserisci una email");
161             return;
162         }
163
164         $.ajax({
165             url: "/api/richiedi_recupera_pw",
166             ...
167         }) . success (function (res) {
168             alert("Ti e' stata inviata una email all'indirizzo
169                 specificato");
170         });
171     }
172
173 </script>
174
175     {% block script %} {% endblock script %}
176
177 </body>
178
179 </html>

```

Il template *base_table* estende *base*, introducendo una *datatable* generica nella sezione *content*.

Listing 28: Template base_table.html

```

1  {% extends 'base.html' %}
2
3  {% block content %}
4  <h1 class="title pb-3">
5      {% block table_title %} Titolo tabella {% endblock table_title %}
6  </h1>
7  <table
8      cellpadding="0"
9      cellspacing="0"
10     border="0"
11     class="dataTable table table-striped"

```

```

12   id="example"
13 ></table>
14 {% endblock content %}
```

A questo punto è possibile creare un template per ogni pagina del sito, il quale implementa la logica dell'applicazione.

Le operazioni implementate in ogni template sono:

1. *Fetch dei dati*, tramite chiamata *ajax*;
2. *Visualizzazione dei dati nella datatable*;
3. *Operazioni sui dati*: create, update, delete, reload.

Listing 29: Esempio template - persona.html

```

1  {% extends 'base_table.html' %}
2
3  {% block table_title %} Membri registrati {% endblock table_title %}
4
5  {% block script %}
6  <script>
7    $(document).ready(function() {
8
9      var qualificaOptions = { "caposquadra" : "Caposquadra" , "
10        volontario" : "Volontario" , "coordinatore" : "
11        Coordinatore" };
12
13      var permessiOptions = { "crea_evento" : "Gestione eventi" , "
14        cms" : "Gestione piattaforma" };
15
16      // valore iniettato dal template engine (lato server) , in
17      // quanto presi dal database
18      var abilitazioniOptions = [{{ abilitazioniOptions | safe
19      }}];
20
21      var columnDefs = [
22        // utilizzato per update , invisibile all 'utente .
23        // Contiene il valore di "numero_tessera" prima della
24        // modifica .
25        {
26          data: "numero_tessera_old" ,
27          visible: false ,
28          searchable: false ,
29          type: "hidden"
30        },
31      ];
```

```
23          {
24              data: "numero_tessera",
25              title: "Tessera",
26              unique: true
27          },
28          {
29              data: "codice_fiscale",
30              title: "Cod. fiscale",
31              unique: true
32          },
33          ...
34          {
35              data: "data_di_nascita",
36              title: "Data di nascita",
37              datetimepicker: { timepicker: false , format : "Y/m
38                  /d" }
39          },
40          {
41              //genera select
42              data: "qualifica",
43              title: "Qualifica",
44              type : "select",
45              options : qualificaOptions ,
46              select2 : { width: "100%" },
47              render: function (data , type , row , meta) {
48                  if (data == null || !(data in qualificaOptions))
49                      return null;
50                  return qualificaOptions [data];
51              }
52          },
53          {
54              //genera multiselect
55              data: "abilitazioni",
56              title: "Abilitazioni",
57              type: "select",
58              options: abilitazioniOptions ,
59              multiple : true,
60              select2 : { width: "100%" },
61              render : function (data , type , row , meta) {
62                  console.log(data)
63                  return data;
64              }
65          },
66          ...
67      ];
68
69      //configurazione datatable
myTable = $('#example').DataTable({
```

```
70      "sPaginationType": "full_numbers",
71      ajax: { //fetch dei dati tramite ajax
72          'url': '/api/get_persona',
73          'type': 'GET',
74          'beforeSend': function (request) {
75              request.setRequestHeader("Authorization", token);
76                  //token autorizzazione
77          },
78          "dataSrc": function ( json ) {
79              json.forEach(function(elem) { //crea copia chiave
80                  elem[ "numero_tessera_old" ] = elem[ "numero_tessera" ]
81                      ];
82              });
83              return json;
84          },
85          error: (response) => {
86              if (response.status == 401){ //se token non valido
87                  showLogin()
88              } else{
89                  alert(response.responseJSON.message) //errore
90                      generico
91              }
92          },
93          columns: columnDefs,
94          dom: 'Bfrtip', // Needs button container
95          select: 'single',
96          responsive: true,
97          altEditor: true, // Enable altEditor
98          buttons: [
99              {
100                  text: 'Nuovo',
101                  name: 'add' // do not change name
102              },
103              {
104                  extend: 'selected', // Bind to Selected row
105                  text: 'Modifica',
106                  name: 'edit' // do not change name
107              },
108              {
109                  extend: 'selected', // Bind to Selected row
110                  text: 'Elimina',
111                  name: 'delete' // do not change name
112              },
113              {
114                  text: 'Aggiorna dati',
115                  name: 'refresh' // do not change name
116              }
117          ],
118          onAddRow: function(datatable, rowdata, success, error) {
119              $.ajax({
```

```
116         url: "/api/create_persona",
117         type: 'POST',
118         dataType: 'json',
119         contentType: 'application/json',
120         data: JSON.stringify(rowdata),
121         beforeSend: function (request) {
122             request.setRequestHeader("Authorization",
123                                     token);
124         },
125         success: function () { rowdata["numero_tessera_old"] =
126                           rowdata["numero_tessera"]; success(rowdata); },
127         error: function(response) { alert(response.
128                                     responseJSON.message) }
129     );
130 },
131 onDeleteRow: function(datatable, rowdata, success, error) {
132     $.ajax({
133         url: "/api/delete_persona",
134         type: 'DELETE',
135         ...
136     },
137     onEditRow: function(datatable, rowdata, success, error) {
138         $.ajax({
139             url: "/api/update_persona",
140             ...
141         });
142     }
143 );
144 });
145 });
146 </script>
147
148 {% endblock script %}
```

Approfondimento: Template con jinja 2

Jinja 2 [?] è un famoso template engine, utilizzato da framework come Flask e Django.

Un *template engine* è una libreria che si occupa di elaborare particolari file, detti *template*, in modo da produrre un documento risultato.

I *template* definiscono una struttura: tramite una particolare sintassi è possibile includere in essi valori, liste o persino altri template.

I template risultano quindi particolarmente utili quando si ha una struttura ricorrente. Questo solitamente è il caso delle pagine web, dove le pagine possono essere composte mediante moduli indipendenti e condivisi.

La sintassi di jinja 2 è molto semplice:

- `{% ... %}` denota uno statement;
- `{{ ... }}` denota un'espressione da stampare nell'output;
- `{# ... #}` denota un commento;
- `#...##` denota un line statement.

Di seguito alcuni esempi per comprendere le direttive principali.

Listing 30: Scrivere il valore di una variabile

```
1 <div> Nome: {{ elemento.nome }} </div>
2 <div> Nome: {{ foo[ 'bar' ] }} </div>
```

Listing 31: Valutare una condizione

```
1 {% if errore %}
2     Errore!
3 {% endif %}
```

Listing 32: Iterare su una lista

```
1 {% for item in seq %}
2     {{ item }}
3 {% endfor %}
```

È anche possibile utilizzare le keyword *extends* e *block* per sfruttare l'ereditarietà in modo da costruire template in modo gerarchico:

Listing 33: Padre

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     {% block head %}
5         <link rel="stylesheet" href="style.css" />
```

```

6      <title>{% block title %}{% endblock %} – My Webpage</title>
7      {% endblock %}
8  </head>
9  <body>
10     <div id="content">{% block content %}{% endblock %}</div>
11     <div id="footer">
12         {% block footer %}
13             &copy; Copyright 2008 by <a href="http://domain.invalid/">
14                 you</a>.
15             {% endblock %}
16     </div>
17 </body>
18 </html>

```

Listing 34: Figlio

```

1  {% extends "base.html" %}
2  {% block title %}Index{% endblock %}
3  {% block head %}
4      {{ super() }}
5      <style type="text/css">
6          .important { color: #336699; }
7      </style>
8  {% endblock %}
9  {% block content %}
10     <h1>Index</h1>
11     <p class="important">
12         Welcome to my awesome homepage.
13     </p>
14  {% endblock %}

```

0.11.4 Testing

Per effettuare il testing durante lo sviluppo, risulta naturale utilizzare il server di test offerto da flask.

Ogni client web mostra i contenuti a suo modo. Sebbene HTML5 sia ormai uno standard de facto, è comunque importante verificare il comportamento del sito web su tutti i browser in commercio. Il team di sviluppo ha verificato il funzionamento del website utilizzando *Google Chrome*, *Firefox* e *Microsoft Edge*, aggiornati ad una versione recente.

La visualizzazione dei conenuti dipende anche dalle dimensioni e risoluzione dello schermo. Per questo motivo è necessario anche effettuare delle prove ridimensionando la finestra e cambiandone la forma.

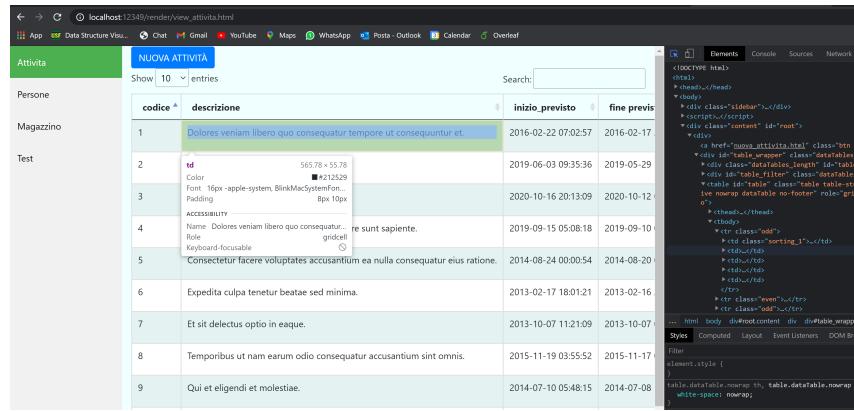


Figura 46: Testing con il browser *Google chrome* durante lo sviluppo del website

0.11.5 Deploy locale

Il deploy è stato effettuato anche questa volta tramite Docker. Sfruttando il fatto che Flask (gunicorn) è già attivo per servire le API, è conveniente includere il website all'interno del container preesistente.

Di seguito alcuni screenshot del risultato finale.

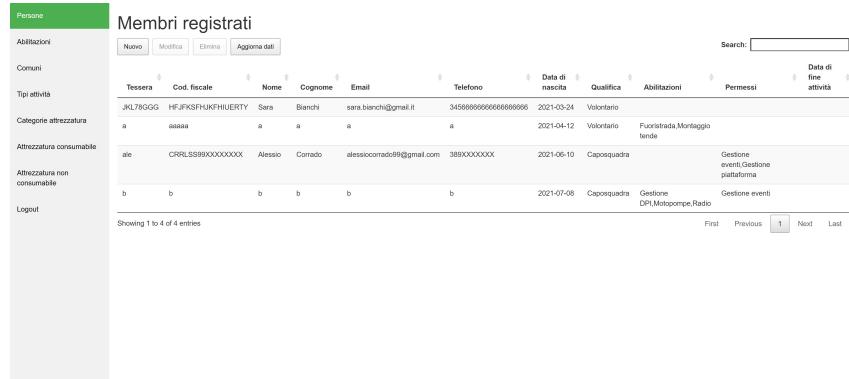


Figura 47: Persona

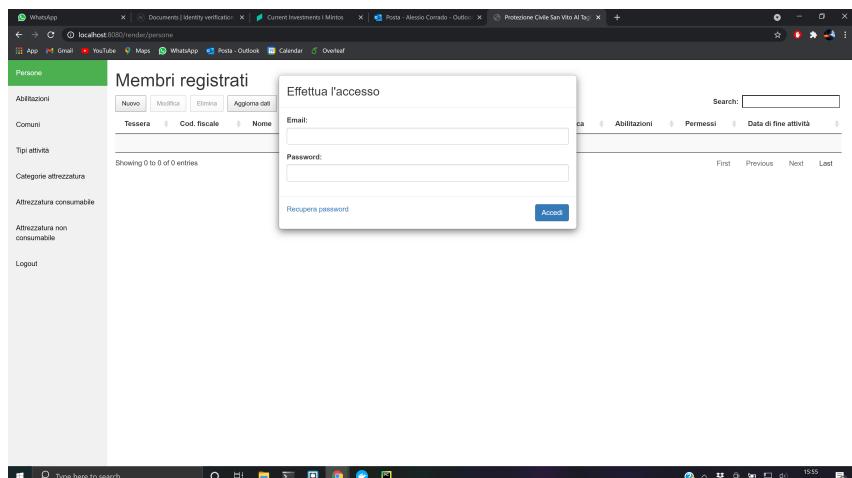


Figura 48: Login

A screenshot of a password recovery form titled 'Recupera password'. It has two input fields: 'Email:' and 'Nuova password:', both with placeholder text. Below the fields is a blue 'Accedi' button.

Figura 49: Recupero password

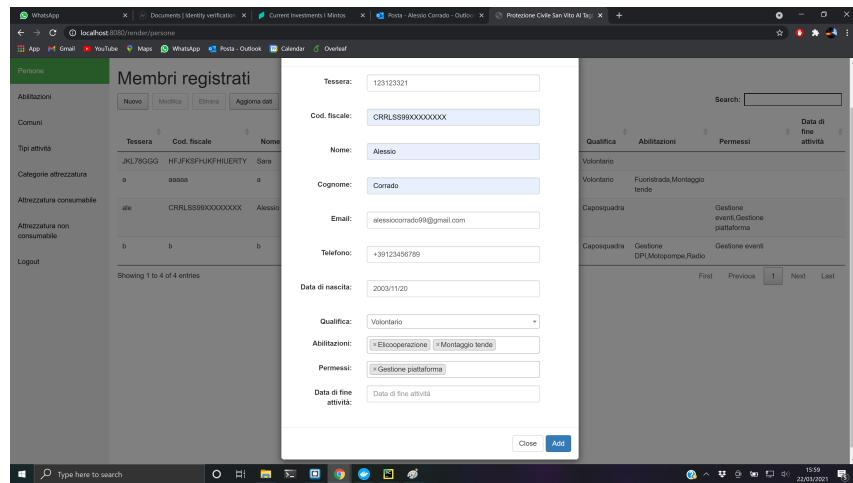


Figura 50: Nuova persona

0.12 Deploy

Per il deploy si è scelto di utilizzare *Google Cloud (GCP)*.

GCP [?] è la piattaforma cloud offerta da Google. Essa contiene una lunga lista di servizi, che riguardano tutti gli aspetti principali dello sviluppo in cloud.

Le principali categorie di servizi sono le seguenti:

- *Computing*: ciò che serve per effettuare computazioni e lanciare istanze di applicazioni;
- *Archiviazione*: file storage persistente;
- *Database*: storage mediante basi di dati, sia relazionali che non;
- *Networking*: gestione delle reti tra servizi e sicurezza delle reti;
- *Big data*: strumenti per lavorare con big data e IoT;
- *Intelligenza artificiale*: modelli preaddestrati e strumenti per l'intelligenza artificiale;
- *Strumenti*: strumenti vari per la costruzione di container, lo scheduling di operazioni, la gestione degli endpoint e altro ancora;
- *Operazioni*: monitoraggio, logging, debugging e profiling dei servizi attivi.

0.12.1 Deploy del database

Per il deploy del database si è scelto di utilizzare *SQL*.

Esso fornisce l'accesso a istanze di database MySQL e PostgreSQL completamente gestite. Ciò vuol dire che è la piattaforma a occuparsi interamente del deploy delle istanze, del loro mantenimento, del load balancing, del backup e tutto il resto.

Considerando il carico ridotto, è stata scelta un'istanza base. Essa dispone di 1 vCPU¹⁹, 3.75 GB di memoria primaria, 10 GB di memoria secondaria HDD.

¹⁹Le vCPU, o *virtual CPU*, sono cpu virtualizzate. Più vCPU possono condividere la stessa CPU fisica.

Un modo semplice per effettuare l'importazione dei dati è collegarsi direttamente all'istanza tramite *shell psql* e importare un dump del database locale.

Per rendere i dati più sicuri, è stato attivato un backup periodico giornaliero.

0.12.2 Deploy delle API e website

Per il deploy delle API e website si è scelto di usare *App Engine*.

App Engine fornisce un servizio di pubblicazione per webapp scritte in diversi linguaggi, tra cui Python.

È anch'esso completamente gestito: la piattaforma si occupa di costruire le istanze docker e caricarle sui server per la pubblicazione. La piattaforma fornisce anche un servizio automatico di gestione del carico, mediante il deploy temporaneo di un maggior numero di istanze.

Per pubblicare una webapp su App Engine è sufficiente richiamare i comandi appropriati tramite shell.

Listing 35: Pubblicazione della webapp su App Engine

```
1 gcloud app deploy
```

Di default la webapp è disponibile ad un indirizzo offerto dalla piattaforma. È possibile acquistare un dominio personalizzato sempre all'interno della piattaforma.

0.12.3 Pubblicazione dell'app

Per pubblicare le app Android e iOS sui relativi store, è necessario costruire delle app standalone, che non siano più legate alla piattaforma Expo.

Per farlo è sufficiente creare un file di configurazione, che istruisca Expo su come effettuare il build, e poi lanciare il comando apposito per la piattaforma.

Struttura del file di configurazione

```
1 {
2   "expo": {
3     "name": "Your App Name",
```

```
4   "icon": "./path/to/your/app-icon.png",
5   "version": "1.0.0",
6   "slug": "your-app-slug",
7   "ios": {
8     "bundleIdentifier": "com.yourcompany.yourappname",
9     "buildNumber": "1.0.0"
10 },
11  "android": {
12    "package": "com.yourcompany.yourappname",
13    "versionCode": 1
14  }
15 }
16 }
```

Comandi per effettuare il build delle due app

```
1 expo build:ios
2
3 expo build:android -t app-bundle
```

Una volta terminato il build e relativi test, è sufficiente accedere al profilo personale degli store e caricare quanto necessario.

0.13 Conclusioni e sviluppi futuri

Svolgere questo progetto è stato particolarmente interessante perché esso racchiude l'applicazione di conoscenze maturate nel corso dell'intero percorso di studi triennale.

Le discipline maggiormente utilizzate sono:

- *Ingegneria del software*: ciclo di vita del software, raccolta ed analisi dei requisiti, diagrammi di progettazione, architettura del sistema, politica di qualità, testing;
- *Interazione uomo-macchina*: design ux/ui, user-centered design;
- *Basi di dati*: sviluppo del database;
- *Reti di calcolatori*: scambio di dati in rete, autenticazione e sicurezza;
- *Programmazione*: strutturazione del codice, implementazione delle soluzioni;
- *Sistemi operativi*: configurazione ed utilizzo di server linux-based.

Il risultato è particolarmente soddisfacente, non tanto per la mera implementazione, ma per essere riusciti a portare avanti un progetto coerente e ben strutturato, nonostante la complessità non indifferente.

Parte del progetto è stato svolto in team. Ciò ha permesso un dialogo costruttivo, in particolare nelle scelte più cruciali in fase di progettazione, ma ha anche mostrato le difficoltà e divergenze del lavoro di squadra.

L'interazione con il committente è stata particolarmente positiva, con una forte propositività e disponibilità.

Allo stato attuale:

- il *database* è implementato, completamente testato e funzionante;
- l'*API* è implementata e funzionante. Necessità di ulteriori test, in particolare riguardo la sicurezza;
- il *website* è implementato e in fase di testing. Necessita della validazione conclusiva da parte del committente e di piccole correzioni di bug;
- l'*app* è completamente progettata ma ancora in fase di sviluppo.

Ci sono stati alcuni ritardi ma, considerando gli impegni universitari e lavorativi dei membri del gruppo, oltre che la natura totalmente gratuita del progetto, il grado di sviluppo raggiunto è valutato soddisfacente da parte di tutti. In un tempo non troppo lungo sarà possibile consegnare al committente una versione completa.

Altri possibili sviluppi, realizzabili a seguito del completamento delle attività, sono:

- Utilizzare lo strumento *query insights* di GCP SQL per ottimizzare le query su database;
- Implementare l'utilizzo di Redis sulle API;
- Progettare ed implementare le funzionalità mancanti dell'app;
- Migliorare la grafica del sito web;
- Effettuare ulteriori test, in particolare per quanto riguarda la sicurezza;
- Eseguire test di profiling per individuare i bottleneck e ridurre i tempi di risposta;
- Progettare test automatizzati per le future release.

Bibliografia

- [1] J. Bišker, *On the elements of the empty set*. Mathematica Absurdica **132** (1999), 13–113.
- [2] U. Pýrlå, *Generalization of Bišker’s theorem*. Paperopolis J. Math. **14** (2001), 125–132.