Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Cross Standard Generator

un generatore di applicazioni web
basato su modello cross

Relatore:
Chiar.mo Prof.
FABIO VITALI

Presentata da:
ANDREA D'ARPA

III Sessione
2018/2019

A Elena e Salvatore...

Indice

Capitolo 1

Introduzione

Lo scopo di questo lavoro è quello di teorizzare un metodo per poter generare in modo automatico applicazioni web basate su operazioni standard su basi di dati, utilizzando un documento di livello comprensibile ad un calcolatore, e contemporaneamente abbastanza astratto da poter essere facilmente interpretabile dall'uomo.
Questo porta alla generazione di una applicazione completa di lato utente e lato server, la cui specifica sia completamente incentrata sull'usabilità.

Nella programmazione web è inevitabile avere a che fare con le basi di dati e dunque risulta fondamentale implementare funzioni di lettura, scrittura, modifica ed eliminazione, ovvero le operazioni CRUD (create, retreave, update e delete). Quello che dunque viene implementato è un server in ascolto di eventuali richieste dagli utenti che, tramite il server stesso, potranno manipolare i dati utilizzando le funzioni appena descritte. Gli utenti per poter compiere queste manipolazioni, necessitano di una interfaccia grafica, che ha lo scopo di rappresentare i dati ricevuti (o da inviare) e le operazioni relative ad essi.
La componente client e la componente server interagiscono tramite le API (Application Programming Interface), che sono un insieme di funzioni di supporto, che per funzionare necessitano di un percorso (path) e di un metodo che specifica il tipo di operazione da intraprendere a scelta tra i metodi get, post, put e delete.
Allo scopo di standardizzare il più possibile l'architettura dell'applicazione, l'applica-

zione web generata automaticamente rispetterà l'architettura REST (REpresentational State Transfer) [Fie00].

Il principio che maggiormente caratterizza un sistema REST, è il principio di uniformità delle interfacce, ovvero il principio che sostiene la regolazione delle modalità di interazione tra i componenti del sistema da parte di convenzioni uniformi.
Per capire meglio, possiamo immaginare un client REST, come un browser web, che è a conoscenza dei percorsi della nostra API, e che "naviga" il server. Quest'ultimo si occupa effettivamente di stabilire il comportamento del client, che si ridurrà dunque solo ad interpretare la rappresentazione delle risorse, e a seguire i percorsi tra una risorsa ed un' altra. Questo comporta la possibilità di poter apportare modifiche al server, senza dover incorrere nel versionamento delle API.
Quando si lavora con le API, è buona norma creare una documentazione del loro funzionamento, ovvero fornire una descrizione in cui viene esplicitato a cosa fa riferimento un path e quale funzione ha quando viene invocato con uno dei metodi. Questa, dovrà essere ben strutturata e semplice da consultare, dato che sarà utilizzata come punto di riferimento durante le fasi di manutenzione e debug del sito.
Infatti, esistono delle tecnologie in grado di generare una buona documentazione in modo piuttosto semplice, come OpenApi, che è una delle tecnologie prese in considerazione. Oltre ad avere come vantaggio il suo formato di scrittura, abbiamo come vantaggio la presenza di soluzioni software che permettono, una volta descritto il modello dell'applicazione con formato OpenApi (che verrà approfondito più avanti), di generare automaticamente una interfaccia che permette di visionare tutte le informazioni per capire la struttura delle API del sito progettato e il loro funzionamento.
Una di queste interfacce è SwaggerHub. Questo tool permette di visualizzare e testare il funzionamento di una specifica di una API tramite una interfaccia grafica, fornita una sua definizione con formato OpenApi.
Nonostante questo rimane il fatto che lo sviluppo di questa tipologia di applicazioni web non abbia uno standard effettivo in termini di scrittura del codice.
Questo dunque porta come diretta conseguenza una probabile scarsa qualità del codice, che si traduce in programmi dalla scarsa manutenibilità, che, all'interno di una realtà

aziendale, si traduce a sua volta in un aumento di costi.
A fronte dei problemi qui elencati, sorge la necessità di individuare un modello di progettazione in grado di standardizzare la struttura di una applicazione di questo tipo e la soluzione proposta sta nel formulare e fornire un processo di generazione automatica partendo da un formalismo in grado di esprimere il funzionamento della applicazione rispettando le regole di un determinato modello di progettazione.

Il modello è stato individuato in CROSS, un semplice modello di progettazione della user experience.
Per concretizzare il processo di generazione automatica, viene dunque proposto uno strumento chiamato Cross Standard Generator (CSG), il cui sviluppo è basato su Crospen Api, una estensione di OpenApi il cui scopo è quello di formalizzare le regole del modello CROSS.
Il lavoro è così articolato:
Nel secondo capitolo vengono analizzati dei generatori di interfacce utente presenti sul mercato da tempo, andando ad analizzare i rapporti tra le interfacce e usabilità del prodotto, analizzando in principio quello che è il processo di sviluppo di un software.
Il terzo capitolo invece indaga il processo orientato agli obiettivi (goal-oriented) CROSS per la creazione di software user-centered. Si puntualizzano qui le differenze con un modello goal-oriented tradizionale, osservando i motivi che hanno portato all'adozione del metodo nella generazione di applicazioni web REST. Vengono analizzate le cause di inusabilità che CROSS vuole contrastare, per poi, dopo una panoramica su CROSS, andare a vedere nel dettaglio quelle che sono le componenti CROSS: Concetti, Ruoli, Operazioni e Strutture e Schemi.
Il quarto capitolo presenta CROSPEN API, il formalismo di rappresentazione e documentazione di Applicazioni Web che rispetta il processo di creazione CROSS. Il seguente formalismo è stato creato da Michel Bellomo nella sua tesi discussa all'università di Bologna nella sessione di laurea di Dicembre 2019 . Il lavoro qui presentato è in effetti considerabile una diretta conseguenza del suo, in quanto la mia tesi si pone di realizzare uno degli sviluppi futuri da lui proposti, ovvero lo strumento di generazione automatica di applicazioni web a partire da una descrizione nel suo linguaggio. Viene dunque ana-

lizzato OpenApi, e successivamente viene spiegata la sua estensione CROSPEN API.
Il quinto capitolo presenta lo strumento in se, raccontando le varie idee di sviluppo, l'implementazione dei concetti e le tecnologie impiegate nella sua realizzazione. Vengono analizzate le quattro componenti dell'applicazione: Crosser (strumento di parsing del file CROSPEN API), Crossback (strumento di generazione del server), DBG (strumento di generazione della base di dati e delle funzioni per la sua gestione) e CrossVue (il generatore di interfacce utente).
Il sesto e ultimo capitolo ha il compito di trarre le conclusioni sul lavoro svolto e di raccontare i propositi di implementazione futuri.

Capitolo 2

Generatori di Interfacce Utente

L'interfaccia grafica rappresenta, dal punto di vista di un utente, l'intera applicazione, poiché lo sguardo di un utente si ferma a questa. In sostanza, una cattiva interfaccia grafica, non intuitiva o anche non gradevole, può portare una fetta di clientela ad abbandonare l'utilizzo del software, cercando alternative meglio presentate.

Lo scopo di CSG, è quello di automatizzare l'intero processo di creazione di una applicazione web, ed essendo il modello da me scelto User-Oriented, l'interfaccia grafica rappresenta la più importante componente dell'applicazione stessa.
Questa, dovrà dunque dare garanzia di usabilità, definita dall'ISO (International Organization for Standardization) come Efficacia, Efficienza e Soddisfazione che un utente ottiene nel raggiungere un obbiettivo in un determinato contesto.
Nel design di una interfaccia grafica, si cerca di avvicinare il modello mentale dell'utente finale a quello del progettista del sistema.

Prima di proseguire ad analizzare nello specifico le metodologie per la generazione delle interfacce e degli esempi di strumenti utilizzati a questo scopo, ritengo sia importante chiarire come il processo di automazione sia rilevante all'interno dello sviluppo di un software.

Lo sviluppo di un software si divide in quattro fasi:

1. La progettazione: la fase in cui vengono definiti gli scopi, gli obbiettivi, i vincoli, le strutture e le tecnologie che saranno poi impiegate nello sviluppo del progetto.

2. L'implementazione: la fase in cui viene effettivamente scritto il software, rispettando le decisioni prese nella fase di progettazione

3. Il test: la fase in cui si eseguono delle operazioni sul prodotto, per verificarne efficienza, correttezza e rispetto dei vincoli

4. La manutenzione: la fase successiva alla pubblicazione del software, dove si implementano ulteriori funzionalità, vengono modificati dei requisiti, e vengono corretti eventuali bug scoperti successivamente al rilascio.

Nella prima fase, per maggiore chiarezza, è necessario redigere la documentazione del progetto. Questa è esprimibile sia con l'utilizzo di formalismi, sia tramite linguaggio naturale. La seconda scelta è sconsigliata, a causa dell'alto grado di ambiguità che il mezzo espressivo presenta. L'utilizzo di formalismi, infatti, garantiscono correttezza, coerenza e consistenza logica. Per questo risultano più vicini ai linguaggi del Calcolatore.
L'utilizzo di formalismi per redigere la documentazione durante la fase di progettazione di un sistema, dunque, ha il vantaggio di disambiguare alcuni dei punti chiave della progettazione stessa, come ad esempio i requisiti. Inoltre, adottare formalismi nella stesura della documentazione, ci dà la possibilità di generare automaticamente task, requisiti e addirittura prototipi.

Quello che questa tesi propone è fare un passo in avanti riguardo a questa automazione, cercando di ridurre il processo di sviluppo di un software (nello specifico della tesi, una applicazione web), portando le fasi di sviluppo a due fasi: la progettazione e la manutenzione.

2.1 Metodi di generazione Formali

Utilizziamo questa tipologia di metodi con due obbiettivi:

1. Disambiguazione: la struttura formale di un linguaggio formale, come già detto in precedenza, non cade vittima di disambiguazione. In questo modo possiamo dunque essere sicuri di cogliere i bisogni degli utenti nei requisiti, e di codificarli al meglio.

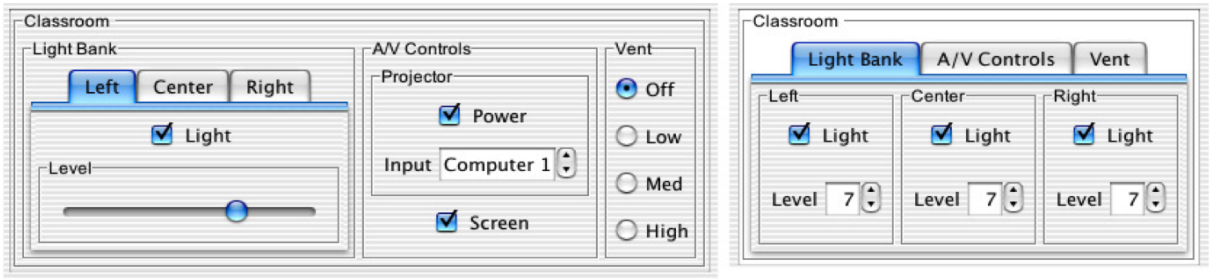
2. Controllo dei requisiti: la documentazione diventa centrale nella verifica del soddisfacimento dei requisiti. Questo può essere di fondamentale importanza in sistemi a tempo reale, dove la sicurezza del sistema è fondamentale.

I vantaggi nell'utilizzo di queste metodologie nella formulazione di una documentazione risultano dunque chiari: diventa semplice individuare gli errori all'interno delle specifiche stesse, si possono spesso validare automaticamente in maniera molto semplice. Inoltre, in molti altri casi si possono anche generare automaticamente dei test, che ci permettono di attuare delle strategie di progettazione test-oriented. È inoltre possibile ottenere dei controlli di qualità in modo automatico. Infine, l'utilizzo di metodi formali nella formulazione delle specifiche ci permette di ottenere una documentazione coerente che riesce a dare supporto a chiunque debba interfacciarsi con questa durante la scrittura del codice. È importante notare che per l'uso di questa metodologia, è presupposto un certo livello di qualificazione del personale, e che il modello deve essere poi rigorosamente seguito per tutto il ciclo di vita del software.
L'uso dei metodi formali, comunque, non presenta solo vantaggi. Salta subito all'occhio il più palese dei difetti nell'uso di questi metodi: la difficoltà nell'uso di questi linguaggi, che richiedono un ragionamento di tipo logico o algebrico. Inoltre, non possiamo dimostrare che un sistema rispetti a pieno le sue specifiche, anche se queste sono espresse in modo formale.
Si deve anche considerare che descrivere in modo specifico delle interfacce utilizzando questi metodi risulta piuttosto complesso a causa della complessità stessa del formalismo (si pensi ad una interfaccia complessa in termini di quantità di elementi a schermo e di possibili interazioni: una descrizione di una interfaccia del genere con un metodo

formale sarebbe difficile da scrivere e ancor più complessa da leggere per un umano).

Il primo caso che andiamo ad analizzare è Supple [Krz04], che mostra il problema in analisi, riducendolo ad un problema di ottimizzazione (nello specifico di soddisfazione dei vincoli)[1]

Il dispositivo che dovrà mostrare l'interfaccia, si occupa di fornire i vincoli, mentre l'ottimizzazione vera e propria avviene sulla serie di operazioni eseguite sulla interfaccia. Il rendering di quest'ultima dunque risulta un oggetto dinamico modificabile. Dunque l'interfaccia deve essere descritta in modo funzionale.
La grande mancanza di Supple (che lo ha portato a non essere più aggiornato dal 2009) è quella di mancare di personalizzazione, caratteristica che una azienda ricerca all'interno di una interfaccia grafica per brandizzare e distinguere il proprio prodotto.



(a) (b)

Figura 2.1: *Un esempio di interfaccia generata con SUPPLE. Dall'esempio dell'interfaccia classroom all'interno di [Krz04]*

Il prossimo caso che andiamo ad analizzare è FILL [WL10], un linguaggio per la modellazione di quella che chiamano logica d'interazione, che considera un'architettura astratta dell'interfaccia a tre livelli che isola nel livello centrale la sola descrizione dell'interazione delle varie azioni.

[1] un CSP può essere definito su un insieme finito di variabili $X_1, X_2, \ldots, X_n$ i cui valori appartengono a domini finiti di definizione $D_1, D_2, \ldots, D_n$ e su un insieme di vincoli (constraints) $C_1, C_2, \ldots, C_n$. Un vincolo su un insieme di variabili è una restrizione dei valori che le variabili possono assumere simultaneamente.

FILL descrive il processo interattivo attraverso delle funzioni definite da input, output e un' eventuale stato di sistema modificato dalle interazioni. Il problema principale in FILL è determinato dalla invisibilità della struttura del sistema. L'utente che si troverà ad utilizzare FILL, utilizzerà dunque l'editor grafico UIEditor.

Continuando la nostra esplorazione dei metodi di generazione di interfacce grafiche, osserviamo ora il caso di Prototype Verification System (PVS) [Mau+17], con la sua interfaccia web PVSio. Questi, oltre a prototipare delle interfacce, aggiunge anche la fase di generazione del codice.
PVS modella il sistema come teoria, utilizzando la logica di ordine superiore, mentre PVSio utilizza un linguaggio grafico basato su automi a stati finiti e la logica per specificare interfacce. L'applicazione permette inoltre di simulare l'interfaccia su vari dispositivi (aggiungendone altri. in caso di necessità). Tramite il suo linguaggio grafico Emucharts, si descrive l'interfaccia come nodi e transazioni, che vengono triggerate da alcuni eventi che si attivano al verificarsi delle relative guardie. Il fulcro di PVS è la generazione del codice in MISRA C, un derivato del C nato nel settore automobilistico. La generazione dunque si riduce ad una traduzione tramite un transpiler da un linguaggio dichiarativo object-oriented ad uno procedurale. Il difetto di PVSio è derivante dalla scarsa versatilità essendo per lo più adatto a lavorare su interfacce con componente fisica come dispositivi medici rendendosi di conseguenza adatta agli attuali sistemi software, sempre piú vicini a dispositivi con interfacce basate sulla virtualizzazione di componenti un tempo fisiche (come smartphone e tablet).

Andando avanti osserviamo il formalismo chiamato Presentational Model, introdotto in (...) in cui si cerca di trovare un metodo per passare da specifiche formali dei requisiti a specifiche informali. L'idea di base è quella di avvicinare collaboratori ad un progetto con skill e background diversi fornendo per l'appunto un linguaggio intermedio. Questo linguaggio aggiunge alla specifica informale, una prima base di formalità, tentando di strutturare le informazioni in componenti tipiche del design human-centered come scenari e casi d'uso. Presentational Model si basa su triple $Widget, Categoria, Comportamento$ mappate a vari identificatori presenti nel contesto del contesto del documento.

Il linguaggio è un'estensione della teoria degli insiemi, con espressione semantica matematica, in modo da ottenere proprietà formali, mantenendo comunque semplicità espressiva.
L'articolo prosegue poi introducendo il concetto di Equivalenza di Design, ovvero generazione concreta di prodotti di design a livello più astratto. Ciò ha lo scopo di aggiungere un altro modo di utilizzare il linguaggio, ovvero verificare se i design dei diversi prototipi dello stesso sistema sono equivalenti.
Tuttavia si evincono due problemi piuttosto evidenti introdotti da questa soluzione: il primo deriva dalla separazione tra specifiche informali e documenti formali, che in seguito all'utilizzo del linguaggio non viene risolta.
Il secondo è invece la staticità delle interfacce descrivibili con Presentational Model, che infatti non tiene conto dei cambiamenti di stato dell'ambiente.
Viene proposta una soluzione dagli sviluppatori per risolvere il problema, che consiste nell'introduzione di automi a stati finiti, che tuttavia va in contraddizione con la semplicità del linguaggio, rendendolo eccessivamente complicato sia dal punto di vista espressivo per gli utenti (ovvero gli sviluppatori che andranno ad utilizzare il metodo) sia dal punto di vista di interazione interna tra gli stati, introducendo un alto livello di complessità nell'interpretazione umana dell'interfaccia.

Possiamo ora trarre una conclusione su queste metodologie: Oltre l'evidente difficoltà di utilizzo al di fuori del mondo accademico, come fatto notare da [Spil14], questi metodi non si focalizzano sull'utente ma sul sistema stesso.

2.1.1 I Metodi Semi-Formali

I metodi semi-formali sono tecniche che utilizzano i linguaggi formali che però non hanno una semantica formalizzata tramite regole logiche o algebriche. È una categoria ben più ampia rispetto a quella dei metodi formali. Questa caratteristica è dovuta dalla varietà di modi in cui un linguaggio semi-formale è configurabile. Infatti, anche solo con l'uso di XML apre le porte alla formulazione di numerose tecniche.
Questi condividono le stesse caratteristiche dei metodi formali per quanto riguarda vantaggi, svantaggi e obbiettivi. Inoltre, anche questi sono orientati verso il processo di

automazione. Infatti, tolti i casi in cui l'implementazione è puramente teorica, abbiamo sempre la presenza di librerie e parser che accompagnano tali metodi.
Prenderemo dunque in considerazione quattro parametri di valutazione presi da [Haa92]: completezza, applicabilità, usabilità e validità.

Andiamo ora ad analizzare in modo più approfondito due categorie di formalismi: uno per la rappresentazione e uno per la trasformazione delle interfacce.

UIDL

[SV03] e [Jov16] La prima che andiamo ad analizzare é UIDL User Interface Descriprion Language. Fanno parte di questa famiglia tecnologie come JavaFX xhe utilizzando FXML e CSS permette la creazione di interfacce grafiche utilizzando anche Scene Builder, il suo editor Drag and Drop.
Nella maggior parte dei casi i linguaggi appartenenti a UIDL sono derivati di XML, e qui ne vediamo un paio di esempi in breve, come nella scorsa sezione.
XUL XML User Interface Language è un linguaggio basato sulla separazione in definizioni, logica e presentazione. L'interfacia viene descritta con oggetti grafici ed attributi, mentre la logica viene implementata con l'utilizzo di script. Il difetto che ha portato alla deprecazione del linguaggio a favore di HTML5, è quello di non adattarsi a schermi di piccole dimensione, mancando di una delle specifiche fondamentali delle interfacce web moderne: il mobile-first.
UIML è un metalinguaggio dichiarativo. Una specifica UIML presenta: una specifica di interfaccia, una sezione di collegamenti esterni e dei template che ha lo scopo di rendere possibile il riuso degli elementi. Il suo difetto preponderante è la mancanza della possibilità di descrivere diverse implementazioni della stessa interfaccia su dispositivi diversi, dato che in UIML ogni documento si riferisce ad una singola piattaforma.
L'ultimo caso di linguaggi per la descrizione delle interfacce che andiamo ad analizzare è Maria. Questo linguaggio, con il suo framework Mariae sviluppato dal CNR[PSS09][MPSS13], è specializzato nella generazione di interfacce multi-modali web. L'idea di base è quel-

la di implementare le proprietà CARE[2]. Dopo una prima specifica XML astratta per descrivere le trasformazioni che specificano l'interazione dell'applicazione sotto forma di condizioni, azioni oppure di eventi, aggiungiamo una seconda specifica definita concreta che descrive il tipo di dispositivo per cui verrà creata l'interfaccia. In questa seconda fase vengono specificati i tipi di widget e gli altri elementi che si occuperanno di concretizzare le specifiche astratte precedentemente esposte.

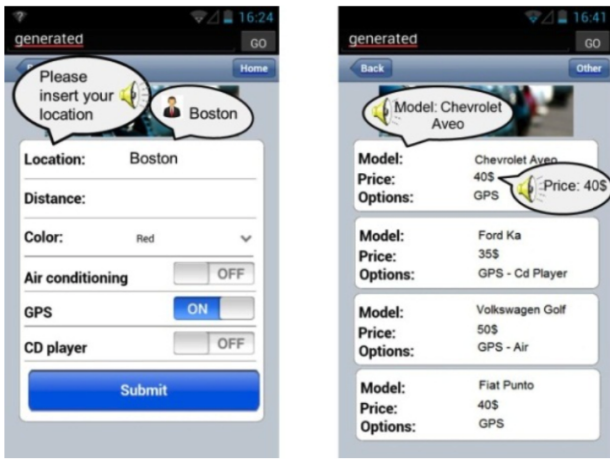


Figura 2.2: *Un esempio di un interfaccia mobile generata con Maria*

In seguito a questo elenco di esempi, possiamo concludere con un paio di osservazioni: le tecnologie qui elencate in molti casi considerano la diversità di ambiente, ma non viene considerato il contesto di utilizzo della interfaccia.

User Interface Trasformation Languages

I linguaggi per la trasformazione delle interfacce, in determinati casi, sono pensati come componente appartenente ad uno stack di tecnologie di design. Allo stesso modo, esistono altri linguaggi appartenenti alla categoria, più general purpose, ma ben predisposti alla coadiuvazione con tecnologie già presenti sul mercato
Il primo esempio che andiamo a vedere è GT ovvero Graph Transformation. Questo presenta delle regole di produzione che utilizzano una parte sinistra ed una parte destra.

[2]Complementary, Assignment, Redundancy and Equivalence

Chiarendo il concetto, si esprimono delle condizioni nella parte sinistra che, quando verificate, portano a ciò che viene descritto nella parte destra.
Questo permette di portare una descrizione di livello astratto a livello concreto, composto da widget e componenti grafiche, come mostrato in [LVM+04] dove viene applicata la tecnica su UsiXML la cui struttura ricorda un grafo.

ATL (Atlas Transformation Language) è invece un formalismo per la trasformazione di diagrammi UML (Unified Modelling Language) [3].
Risulta importante per la possibilitá che fornisce di scegliere tra uno stile di scrittura puramente dichiarativo basato su pattern matching, ed uno stile con componenti imperative e componenti dichiarative.

2.2 Conclusioni sulle tecnologie preesistenti

Le limitazioni presentate dai linguaggi preesistenti qui elencati hanno sicuramente portato alla ricerca di una soluzione diversa nella scelta di un linguaggio per dichiarare i requisiti e per disegnare l'applicazione.
Nonostante quasi tutte fossero compatibili con un processo di generazione automatica, il linguaggio che viene ricercato per implementare il software CSG, non può essere individuata tra quelle qui esposte per tre motivi:

1. Non abbiamo in nessuno dei casi uno standard effettivo di progettazione nella formulazione dei vincoli e nel processo di produzione, fondamentale nella proposizione di un possibile standard

2. Nessuna delle soluzioni proposte presenta equilibrio tra semplicità espositiva, e completezza.

[3] UML è un linguaggio dalla notazione semi-formale e semi-grafica atta alla descrizione di soluzioni analitiche e progettuali sintetiche e facilmente comprensibili.

3. Le interfacce così generate risultano essere troppo "statiche" in alcuni casi. Le alternative che non presentano questo difetto sono invece o legate a tecnologie specifiche, oppure troppo generiche.

Nel prossimo capitolo sarà individuato lo standard da adottare per risolvere il problema esposto al primo punto, e successivamente, andremo ad analizzare il linguaggio che lo formalizza, che sarà utilizzato come base per l'implementazione del tool.

Capitolo 3

CROSS: Concetti Ruoli e Operazioni creano Strutture e Schemi

Una delle più grandi mancanze delle tecniche formali di documentazione e costruzione del software è la mancanza di guide alle pratiche che si vogliono formalizzare. Proprio per questo, queste tecniche non vengono adoperate all'interno di contesti lavorativi dove si possono spesso trovare persone non direttamente qualificate a svolgere il ruolo nel settore informatico.
Inoltre, c'è da considerare che l'usabilità è da poco considerata all'interno delle realtà aziendali, ed essendo stata in passato una realtà al più accademica, nei laureati che lavorano nel campo si può riscontrare una perdita di coscienza riguardo le metodologie user-centered.
Infine, anche la componente economica gioca un ruolo fondamentale nelle limitazioni che portano un prodotto alla distanza con il modello user-centered. Si pensi infatti, ad esempio, a realtà più modeste, dove il budget porta a risultati equiparabili ad applicazioni, e nello specifico di questa tesi applicazioni web, piuttosto arretrate.
CROSS (inizialmente "CAO=S"), è una semplificazione di approccio di design goal-oriented in progetti in cui il budget non permette un'analisi sistematica dell'utenza destinataria ne tanto meno il coinvolgimento di esperti di usabilità esterni. Si pensa dunque ad una serie di task diversificati a seconda dei tipi di utente, che vengono rappresentati tramite delle caratteristiche determinate a priori.

3.1 Le cause dell'inusabilità

Andiamo ora ad analizzare le cause di inusabilità.

Distanza tra modello espresso dal sistema e modello atteso dall'utente

Sono legate a questa categoria i problemi di linguaggio utilizzato, che può risultare distante dagli scopi di contesto del sito, o più semplicemente l'utilizzo di un linguaggio non adatto al tipo di utenza di destinazione, l'assenza di suggerimenti di spiegazioni non abbastanza chiare oppure degli output con scarsità di testo.

Complicazione procedurale

I task dell'utente possono essere concepiti in maniera più complessa del necessario, allontanandosi da un flusso di operazioni realistiche, ad esempio richiedendo una serie troppo lunga di passaggi operazionali. È altrettanto problematica la situazione in cui i singoli passaggi portino ad un carico di informazioni troppo pesante per l'utente.

Incompletezza delle viste

Le informazioni visualizzate nelle schermate possono essere troppe o troppe poche, oppure le informazioni utili per un task potrebbero essere divise in più schermate.

3.2 Panoramica su CROSS

CROSS si focalizza per lo più sulla fase di analisi, prendendo in analisi un pool di utenti finali da utilizzare come parametro per task e obbiettivi. Risulta dunque fondamentale che questi due parametri vengano analizzati ed approfonditi nel modo più accurato possibile, concentrandosi sulle caratteristiche che hanno un effettivo impatto sull'applicazione. Il metodo stesso, incoraggia l'uso di design pattern o di pattern grafici. Incoraggia anche all'uso di tool per la generazione automatica di prototipi dell'interfaccia a partire dalla documentazione di analisi dei ruoli, delle operazioni e dei concetti. Questo ad ogni modo non esclude una documentazione informale da accompagnare ad eventuali documentazioni formali atte alla definizione specifica.

Risulta a questo punto chiaro il motivo principale per cui si è scelto CROSS per porre le basi del nostro strumento di generazione: si vuole soddisfare la richiesta del modello stesso di avere dei tool specifici per la generazione di interfacce e prototipi.
Andiamo ora a comprendere come CROSS svolga la fase di analisi dei requisiti, svolta in maniera leggermente diversa rispetto all'analisi tradizionale. Abbiamo tre fasi: raccolta, analisi e rappresentazione dei requisiti.
La fase di rappresentazione ha come scopo fondamentale la divisione dei requisiti in due categorie: funzionali e non funzionali (come i requisiti utente) e la loro disambiguazione.
L'utente è centrale nel processo di produzione CROSS. Per essere più chiari, la sua centralità è derivante dai ruoli.
Quindi nella fase di analisi, si lavora sui concetti, così come vengono visti da ogni ruolo. Non avremo più delle specifiche in termini tecnici (funzioni e strutture di dati) ma in termini di Operazioni e Concetti.

3.3 Le componenti CROSS

Procediamo ora a vedere nel dettaglio le componenti del modello CROSS. Si avvisa il lettore che questo paragrafo ha l'obbiettivo di dare una visione semplice e di insieme delle varie componenti, senza cadere nello specifico.

3.3.1 I Concetti

Rappresentano le informazioni su cui lavorerà il sistema. Sono la visione dell'utente sugli oggetti manipolabili, le componenti base che fanno percepire il sistema all'utente finale.
L'utente, avendo una visione astratta del sistema, percepisce l'interazione con il sistema come una serie di operazioni generali, e non come delle manipolazioni di stato per mezzo di funzioni. Questo vuol dire, che il concetto di transazione (se presente) dovrà essere trasparente all'utente finale del sistema.
I concetti, si possono ricavare dall'osservazione dell'utente. Se questo non fosse possibile, rimane comunque possibile fare uso della documentazione grezza scritta durante la stesura dei requisiti.

La scelta dei termini per i concetti(aggettivi o sostantivi) è un'operazione delicata, durante la quale è fondamentale tenere a mente alcune accortezze:

1. fare attenzione a quando ruoli e programmatori usano nomi diversi per lo stesso concetto
2. fare attenzione a quando ruoli diversi usano nomi diversi per lo stesso concetto
3. fare attenzione a quando ruoli diversi fanno uso della stessa parola per descrivere concetti diversi
4. fare attenzione alle parole che possono assumere significati diversi anche nello stesso ruolo

CROSS fornisce dei metodi di disambiguazione. Rispettivamente

1. Si da priorità ai termini che utilizzano i ruoli rispetto a quelli utilizzati dai programmatori, evitando acronimi e abbreviazioni non spiegati.
2. Si cercano dei termini che siano compresi allo stesso modo per tutti i ruoli oppure, se ciò non è possibile, si utilizza un termine diverso per ogni concetto.
3. Si cercano dei termini diversi, che siano sinonimi o espressioni, per sostituire i termini ambigui
4. (come il terzo punto)

È molto importante non usare mai un verbo per esprimere un concetto in CROSS. In caso dovesse accadere si cerca una espressione metaforica del concetto stesso. Si prenda ad esempio "leggere le mail" che diventa la "casella di posta".
Durante la progettazione, in fine, è importante tenere conto sin dalle prime fasi della coerenza. Ogni modello dovrà dunque essere legato ad un ruolo che lo crea. Anche l'eliminazione va specificata in un certo modo, dando il permesso di eliminare un concetto solo a ruoli che ne hanno motivo.

3.3.2 I Ruoli

Sono coloro che in ingegneria del software vengono definiti Stakeholder[1].
In CROSS siamo interessati proprio a questa categoria, che ha la possibilità di dare dei vincoli e impattare la progettazione, in cui possiamo trovare figure come committenti, vincoli legali, finanziatori e anche utenti finali.
Nei metodi orientati agli obiettivi (goal-oriented), solitamente si fa uso di personas[2][] che ha come prerequisito una ricerca etnografica.
In CROSS, il processo è simile, ma ben più semplice. Si definisce infatti un personaggio come l'insieme di sei caratteristiche decise a priori, che hanno importanza e impatto sull'usabilità del sistema. Inoltre, ci si limita a costruire un numero minore di personaggi che in CROSS diventano un paio per ruolo.
In CROSS ci chiediamo per quale target stiamo progettando il sistema, e dunque, ci chiediamo per quale livello delle varie caratteristiche ha senso progettare il sistema.
Le caratteristiche hanno una valutazione numerica su una scala che va da 1 a 5, e con i valori ricavati si costruirà un diagramma. Le caratteristiche che andiamo a valutare sono:

1. Competenza Tecnica: capacità di utilizzo di strumenti informatici, non intesi come computer in senso stretto, comprendendo smart device vari su cui l'applicazione potrà essere utilizzata. Un caso limite si può trovare ad esempio in una persona che utilizza un solo tipo di dispositivo (come ad esempio uno smartphone).

2. Competenza di Dominio: quantità di nozioni conosciute pratiche e teoriche riguardanti l'ambito in cui si opera, comprendente di conoscenza di linguaggio tecnico-specifico.

[1] Tutti i soggetti, individui od organizzazioni, attivamente coinvolti in un'iniziativa economica (progetto, azienda), il cui interesse é negativamente o positivamente influenzato dal risultato dell'esecuzione, o dall'andamento, dell'iniziativa e la cui azione o reazione a sua volta influenza le fasi o il completamento di un progetto o il destino di un'organizzazione.
da "Enciclopedia Treccani"

[2] tecnica che consiste nella creazione di personaggi dal quale poi scegliere un personaggio principale e dei secondari. I personaggi scelti devono essere di interesse, e vengono presi in considerazione dal team di sviluppo durante l'intero lavoro, al fine di orientarsi nelle scelte e ricordare il destinatario del progetto.

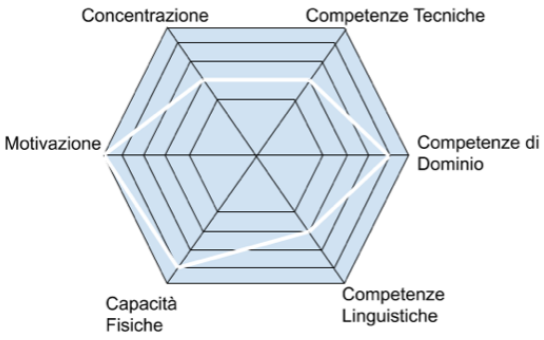
3. Competenza Linguistica: conoscenza della lingua che il software andrà ad utilizzare

4. Capacità Fisiche: presenza di limitazioni fisiche o mentali che possono affliggere l'uso del sistema.

5. Motivazione: l'insieme di motivi per cui un utente dovrebbe rivolgersi allo strumento fornito per completare un task.

6. Concentrazione: qualità dell'ambiente e del contesto d'uso in cui l'utente usa il software.

Analizzando i valori ottenuti, costruiamo un grafico a radar con assi in cui i valori vanno dal punto più esterno con valore 5 al punto più interno di valore 1. Possiamo così isolare l'area descritta dai punti dei vari valori, ottenendo la così detta area di strategia, che indica quali sono le zone su cui lavorare.
Più il valore è alto, più è importante focalizzarsi sullo sviluppo di una caratteristica all'interno del sistema.
Il personaggio creato, quindi, deve essere realistico: non perfetto, ne eccessivamente problematico. Dall'analisi di questo, possiamo dunque ottenere delle valutazioni di caratteristiche rilevanti per la costruzione del sistema.
Si puntualizza che il personaggio in CROSS fa uso anche di una tecnica appartenente al dominio delle tecniche goal-oriented: lo scenario, che non è altro che una descrizione di un evento di utilizzo del sistema da parte di un personaggio.
Mostriamo dunque una scheda di un personaggio per il modello CROSS. Prendiamo come esempio un utente di un sistema di acquisto online di strumenti musicali.

	
John Doe, 45 anni, Londra, Sposato, 1 figlio.	Dopo il lavoro, torna a casa per passare il tempo con sua moglie e suo figlio. Usa il computer in ufficio per motivi di lavoro, ma quando torna a casa preferisce utilizzare il suo iPad. Si dedica alle sue attività di sera, dopo aver messo suo figlio a letto e aver terminato tutti i suoi doveri. Suona la chitarra da quando ha 15 anni, in una accademia di musica classica. Acquista regolarmente online corde per chitarra classica, e una volta l'anno si concede l'acquisto di uno strumento musicale diverso
SCENARIO 1: John ha rotto il mi cantino della sua chitarra classica, e dunque decide di acquistare un nuovo set di corde. Riesce ad acquistare immediatamente le sue corde preferite, che sono mostrate nella home del sito come preferito, tramite la funzione di acquisto rapido, non dovendo passare per il carrello, ne dovendo inserire le informazioni della carta di credito	SCENARIO 2: John ha ricevuto un aumento, e vuole acquistare un set per registrare la sua musica. Selezionando dal menu di categoria laterale, sceglie la categoria Home Studio e Dj Equipment, e filtra i risultati prima sulle schede audio, e poi sui microfoni, ordinando i risultati per prezzo. Finisce per scegliere in entrambi i casi l'articolo meno costoso che ha un feed utenti di 5 punti su 6

3.3.3 Le Operazioni

Le operazioni sono le azioni eseguite dagli utenti sui concetti. L'utente finale dell'applicazione ha la percezione di modificare e lavorare con concetti e non dati, che ricordiamo essere invisibili all'utente. La componente visiva dunque non deve mai mostrare dati riconducibili all'implementazione, ma deve sempre e solo fare riferimento ai concetti con cui il ruolo che ha l'utente che sta interagendo con il software può visualizzare.
Le operazioni corrispondono alle operazioni dei modelli CRUD e REST: Creazione, Lettura, Modifica ed Eliminazione.
Nella progettazione dei task e delle operazioni bisogna tenere conto di possibili valori di default di un concetto, distinzione tra concetti plurali e singolari, l'eventuale persistenza dei dati...
Quest'ultima, può essere gestita da alcuni ruoli, dal sistema in maniera automatica, o dall'utente stesso.
La singola operazione, non va accomunata ad una singola funzione o procedura, in quanto spesso, ad una singola operazione corrisponde una serie di funzioni e procedure per arrivare al risultato.
Bisogna inoltre considerare la quantità di esecuzioni dell'operazione, così come la quantità di ruoli che hanno ad essa accesso.
Vediamo ora in breve le quattro operazioni, osservando cosa tenere a mente rispetto ai ruoli durante la loro progettazione.

Creazione

È la creazione di una nuova istanza nello stato e, se presente nel sistema, nella base di dati. Può essere eseguita manualmente dall'utente, oppure essere gestita automaticamente dal sistema come conseguenza di un'altra operazione. Spesso ha dei valori di Default suggeriti per facilitare la creazione della istanza e per rendere questa operazione meno ripetitiva. È possibile anche tenere conto della storia dell'utente, suggerendo valori usati frequentemente ad esempio. Deve dare un feedback con un messaggio per comunicare il successo o il fallimento dell'operazione.

Aggiornamento

È una operazione che comporta la modifica di un'intera istanza di un oggetto o di alcune sue componenti. Oltre alle accortezze grafiche, ha senso considerare e gestire casi come il cambiamento dello stesso campo in più oggetti, oppure una variazione di un campo a seguito di un evento o di una scadenza temporale.

Vista

È l'operazione che consente di visualizzare a schermo le informazioni richieste. Deve adattarsi al tipo di richiesta, e per questo, presentare modalità differenti per gestire l'ammontare di informazioni da visualizzare. Questo concetto verrà approfondito più avanti, quando si parlerà di generazione di interfacce in CSG.

Rimozione

È l'eliminazione di un concetto dalla vista dell'utente e/o dalla memoria del sistema stesso. L'operazione di rimozione ha due macro: l'eliminazione, che cancella in modo definitivo una istanza dal sistema, eliminando dunque la sua presenza anche dalla base di dati, e l'archiviazione, che rende l'accesso all'istanza archiviata possibile, ma la sposta in un archivio apposito contenente tutte le altre istanze archiviate. Quando viene eseguita una operazione di rimozione, è necessario fornire, come nella creazione, un feedback all'utente per comunicare un successo o un fallimento. In caso di archiviazione, il feedback dovrà inoltre fornire delle informazioni riguardanti i metodi di accesso alla risorsa archiviata.

3.3.4 Strutture e Schemi

L'ultima componente di CROSS, e sono l'output della sua applicazione su concetti e ruoli (si ricorda l'acronimo CROSS sta per Concetti Ruoli e Operazioni producono Strutture e Schemi).
L'output è formato da strutture dati e di navigazione e da schemi, o meglio le viste del sistema. Lo schema è composto dalla vista, concettualmente vicina alla vista della struttura dati, e dai comandi relativi alle operazioni o alla navigazione.

Si parte da un costrutto che viene chiamato diagramma principale, che ci dice in che modo una operazione o è concessa ad un ruolo r su un concetto c. Da qui si può costruire l'insieme di viste, tenendo conto dei vincoli espressi, che permettono di eseguire le operazioni espresse nel diagramma principale. Dopodiché si passa alla costruzione del diagramma di navigazione.
Per evitare errori nella costruzione di strutture e schemi bisogna fare in modo che ad ogni concetto corrisponda almeno un ruolo creatore ed uno che possa visualizzarlo. Dobbiamo inoltre evitare che ci siano parti del diagramma vuote non motivate, porre dei limiti su ciò che è possibile cambiare in un oggetto anche in caso si abbiano i permessi di accesso e, infine, ogni oggetto deve avere la possibilità di essere eliminato.
La creazione dell'applicazione, partendo dal diagramma prodotto, può considerarsi semi algoritmica.

Capitolo 4

CROSPEN API

CROSPEN API è un'estensione di OpenApi 3.0 ideata da Michel Bellomo che ha lo scopo di fornire una specifica al modello CROSS, che altrimenti rimarrebbe un metodo astratto, fatta eccezione dei diagrammi prodotti.
La scelta è dovuta alla disambiguità del linguaggio CROSPEN, e dalla compatibilità totale con strumenti di validazione [Smaa].
La creazione di documentazione formale tramite l'utilizzo del formalismo CROSPEN, ovviamente, non prescinde dalla creazione di documentazione informale, che risulta triviale in particolar modo nella fase di analisi iniziale.
Per garantire dunque un buon grado di flessibilità d'uso ed evitare barriere di apprendimento, CROSPEN è stato progettato come metodo semi-formale, garantendo dunque una alto grado di leggibilità sia da parte di esseri umani, che da parte delle macchine.

4.1 OpenApi

OpenApi è uno standard de facto per la descrizione di applicazioni web REST. Nelle sue prime versioni era riconosciuto sotto il nome di Swagger. È di proprietà di SmartBear Software, tuttavia il progetto in se è di natura OpenSource in collaborazione con Linux Foundation.
Un file di specifica OpenApi è scritto in JSON o in YAML, e sono composti da una serie di oggetti standard. Considereremo in questo paragrafo come estensione di file il

linguaggio YAML (YAML Ain't a Markup Language).
La scelta risiede nel più alto grado di leggibilità (all'uomo) fornita dal linguaggio. Infatti, non solo ha lo stesso potere espressivo di JSON, ma è un suo super-insieme. È definibile come un linguaggio per la serializzazione di dati, e la sua natura, così come JSON, è quella di memorizzare dati.
Una delle caratteristiche che ha favorito il successo di OpenApi è la presenza di librerie e applicazioni che permettono la generazione di documentazione standard, validazioni e tool per il automatico alla versione successiva.
Esistono inoltre strumenti per generare documentazione OpenApi a partire da altri linguaggi di programmazione, rendendo la stesura della documentazione parte integrante della stesura del codice. Si possono inoltre generare dei test (non di usabilità) a patto che sia presente una interfaccia o almeno un suo prototipo.
Ad esempio uno strumento di supporto a OpenApi è Swagger Hub, una piattaforma contenente una serie di tool per validare e generare documenti OpenApi, e generare interfacce di documentazione e test del funzionamento delle Api.

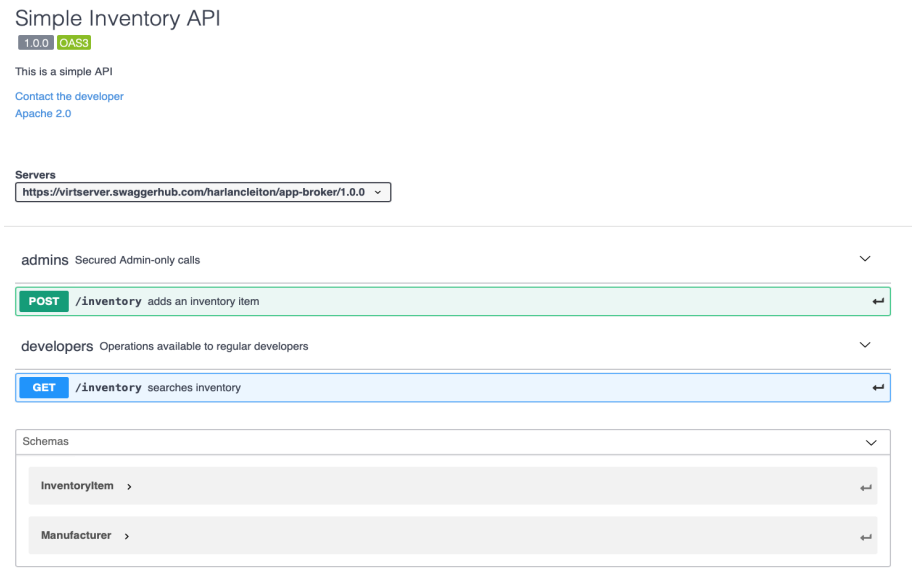


Figura 4.1: *Documentazione generata da Swagger Hub*

4.1.1 Struttura di File OpenApi

Prendiamo ora in esempio un file OpenApi in formato YAML proveniente da [OAI].

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: A paged array of pets
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pets"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
    post:
      summary: Create a pet
      operationId: createPets
      tags:
        - pets
      responses:
        '201':
          description: Null response
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
```

```yaml
  /pets/{petId}:
    get:
      summary: Info for a specific pet
      operationId: showPetById
      tags:
        - pets
      parameters:
        - name: petId
          in: path
          required: true
          description: The id of the pet to retrieve
          schema:
            type: string
      responses:
        '200':
          description: Expected response to a valid request
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pet"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
components:
  schemas:
    Pet:
      type: object
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
    Pets:
      type: array
      items:
        $ref: "#/components/schemas/Pet"
    Error:
      type: object
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
```

Partendo dall'alto, troviamo gli oggetti **openapi** e **info**. Questi mostrano la versione dello standard adottata nel documento e le informazioni essenziali su di esso. Inoltre, anche se nell'esempio mostrato non è presente, può essere all'interno dell'oggetto info

un oggetto description, il cui scopo è quello di dare descrizioni informali. Si precisa, che l'oggetto description è ritrovabile all'interno di vari altri oggetti.

Successivamente troviamo l'oggetto **servers**. Questo ha la funzione di specificare uno o più server che appariranno nella prima parte degli indirizzi delle richieste.

Andando avanti troviamo l'oggetto **path**, che è il più ricco di informazioni. Nell'architettura REST i path (ovvero i percorsi) individuano in modo univoco le risorse centrali. Quindi, nell'oggetto path, descriviamo i diversi punti di dialogo (bidirezionale) tra client e server. Questo serve a mostrare il modello logico dell'applicazione come viene visto dall'utente.
Al suo interno possiamo trovare altri oggetti e attributi. L'oggetto più importante è l'oggetto **operation** che deve apparire in ogni path. Gli oggetti operation sono **get, post, put** e **delete**, e rappresentano le operazioni nell'architettura REST. All'interno di una operation, possiamo trovare altri oggetti come **parameters** e **responses**, che come dice il nome stesso degli oggetti, descrivono rispettivamente i possibili parametri all'interno dell'indirizzo e il tipo di risposte che possono essere restituite dall'esecuzione dell'operazione.
È possibile inoltre trovare un oggetto **schema**, che permette di descrivere i tipi di dati che sono passati in input o in output durante il dialogo.

L'oggetto **tags** può apparire sia alla radice del file, sia dentro un oggetto path. Nel primo caso ha il compito di elencare le categorie utilizzate per creare una gerarchia tra i path e le operazioni. Nel secondo caso, invece, può essere trovato nello specifico all'interno di una operazione, indicando dunque a quale categoria indicata a appartiene.

Vediamo infine l'oggetto **components** che permette il riuso di parti del documento, utilizzando una sua unica definizione. In questo modo si evitano ripetizioni non necessarie, e velocizzando il processo di modifica di un oggetto. È possibile fare riferimento ad un component tramite l'oggetto **reference** utilizzando la seguente sintassi: **$ref: /components/indirizzo/del/file**.

Possiamo trovare all'interno dei components oggetti di tipo schemas, responses, parameters, examples, links, callbacks, requestBody, headers e securitySchemes.

4.2 CROSPEN

Crospen Api nasce, come già accennato, come estensione del linguaggio OpenApi appena approfondito. Crospen è definibile come una versione di OpenApi con una serie di oggetti personalizzati pensati per implementare il modello CROSS all'interno dello standard OpenApi. La sintassi generale di Crospen Api corrisponde a quella di OpenApi, in modo da rendere semplice la migrazione dallo standard già ampiamente in uso alla sua espansione. Inoltre, non modificandone la sintassi, Crospen può usufruire dei vari strumenti di generazione e validazione forniti per OpenApi.
Andiamo dunque a verificare come Crospen aggiunga gli oggetti CROSS a OpenApi. Per i ruoli, Crospen, sceglie di utilizzare dei casi particolari negli oggetti **tags**, che saranno distinti da quelli già presenti in OpenApi, perché preceduti dal simbolo tilde (textbf). Ciò permette di distinguere i **tag** dei **ruoli**, lasciando la possibilità di rispettare la funzione originale dell'oggetto per qual si voglia motivo.
Inoltre, viene fornito anche un metodo di descrizione approfondita per descrivere i personaggi (come si è già visto, figure fondamentali in CROSS) tramite l'utilizzo di un oggetto "**x-characters**". Questi risulta piuttosto ridondante e, come già visto, le descrizioni dei personaggi sono funzionali in fase di analisi, più che in fase di progettazione. Tuttavia, per la natura documentativa del file, è naturale trovare la possibilità di fornire anche delle descrizioni di personaggi.
Mostriamo dunque un esempio per chiarezza di un tag Ruolo in Crospen, prendendo in esame un sito di compravendita di oggetti di elettronica.

```
tags:
        - name: ~Cliente
          description: Chiunque sia registrato

          x-charactes:
```

```
- name: Mario Rossi
identikit: Ragioniere di 34 anni...
domain: 3
technical: 4
language: 5
phisical: 3
motivation: 4
focus: 2
scenarios:
    - Mario vuole comprare un Notebook di
      seconda mano...
    - Mario vuole vendere un vecchio Tablet...
```

In OpenApi i concetti e le operazioni risultano già presenti. Per conformare le specifiche al processo CROSS, in Crospen, troviamo delle aggiunte.
Le operazioni, essendo sia nell'architettura REST che nel modello di progettazione CROSS basate su operazioni CRUD, risultano le stesse con differenze in denominazione. Dunque rimane invariato l'uso dei **tag** (nel nostro caso di **ruolo**) per raggruppare le operazioni sotto diversi gruppi. Se un'operazione è utilizzabile da più ruoli in modo differente, spieghiamo le differenze nel campo **description**.
Si ricorda che se un ruolo non è tra i **tag** descritti dall'**operazione**, prova a chiamarla, questa restituirà un codice HTTP di errore con annesso messaggio. Dunque, nelle **operation** possiamo trovare in aggiunta alla specifica di OpenApi:

post

Presenta 4 nuovi campi:
"**x-auto**", un booleano che indica se l'inserimento di una istanza può avvenire in modo automatico. Può essere seguito da un evento.
"**x-quantity**" é un intero e rappresenta il numero di istanze che si vogliono inserire.
"**x-default**" é un campo opzionale e puó valere "none" oppure una lista di valori o di oggetti schema coerenti con i parametri rispetto all'operazione per ordine e per tipo.
"**x-persistent**" infine é un campo obbligatorio booleano che avverte dell'inserimento

all'interno del database in seguito all'operazione.

put

Risulta poco variato rispetto all'originale, con "**x-auto**" e "**x-quantity**" (identiche in comportamento all'operazione di post) come uniche aggiunte.

get

Rispetto allo standard OpenApi, aggiunge solo un campo: "**x-collection-case**", usato solo in caso ci siano più viste, che può assumere i valori "**list**", "**lookup**" o "**summary**". Nel caso in cui piú ruoli volessero utilizzare una stessa operazione get ottenendo dunque risultati diversi. L'oggetto **responses**, con il quale potremmo fornire le nostre differenze, purtroppo non accetta ridondanza di informazioni. Di conseguenza, possiamo descrivere questo caso solo in modo informale.

delete

Anch'esso ha solo due estensioni: "**x-auto**" analogo a quello sopra esposto, e un nuovo campo obbligatorio "**x-manner**" che accetta solo i valori "**permanent**" o "**archiviation**" sotto forma di lista.

Andando avanti, troviamo l'oggetto "**x-navigation**" che vuole semplicemente implementare la specifica di CROSS di navigazione del sistema in quanto struttura. Inoltre si propone come standardizzazione degli accessori grafici.
Al suo interno ci sono i vari entry-point dell'applicazione. Partendo dal punto di ingresso dunque (al massimo uno per ogni ruolo), ci si estende in profondità. Ad ogni livello si inseriscono oggetti nuovi, che possono essere viste di pagine o intere pagine.
Le nuove pagine devono avere un proprio id nel campo omonimo, possono avere una descrizione (consigliata ai fini della documentazione), un campo **content**, che indica

che l'entità è parte dell'oggetto genitore. Inoltre viene introdotto in "**x-navigation**" l'oggetto "**x-link**" che serve a sostituire il concetto di **$ref** bypassandone le limitazioni.

Capitolo 5

Cross Standard Generator

Arrivati a questo punto, abbiamo analizzato il linguaggio scelto per descrivere la nostra applicazione, e possiamo dunque passare a discutere delle tecnologie impiegate per implementare il tool di generazione automatica di applicazioni web CSG.
In questo capitolo, saranno elencate le scelte progettuali e le proposte implementative del software di generazione automatica, analizzando per ogni suo modulo e componente quali tecnologie preesistenti sono state scelte e come interagiscono con CSG.
È importante sottolineare il concetto alla base di CSG, che non ha lo scopo di eliminare la figura del programmatore per la costruzione di siti basati su architettura REST, ma al contrario, vuole porsi come uno strumento per semplificare il processo di implementazione delle API, proponendo a questo scopo delle soluzioni standard, ed evitando allo sviluppatore stesso tediose sessioni di scrittura del codice ripetitive e frustranti. È ovvia la presenza di una riduzione dei costi di realizzazione da parte dell'impresa nella realizzazione del software stesso.
L'applicazione sarà disponibile sul web con una sua semplice interfaccia interattiva, dove dopo un log in, verrà chiesto all'utente di caricare il suo file con documentazione Crospen Api scritto con estensione YAML (come da specifica di Crospen stesso).
A questo punto verrà generata una cartella compressa da scaricare, con all'interno la documentazione Crospen e il codice generato.
All'utente dunque non rimane che scaricare il codice, installare i pacchetti npm sul proprio server (procedura che sarà guidata da un file README.MD generato anch'esso

da CSG, contenente tutte le istruzioni del caso) e caricare i vari file sul server per poi farla partire utilizzando i comandi di interfaccia node. L'applicazione web generata automaticamente da CSG sarà dunque pronta all'esecuzione e alle modifiche di interfaccia necessarie.
Detto questo entriamo nel vivo dell'applicazione fornendo una breve panoramica sulle tecnologie usate.
Tolta l'ormai chiara scelta del linguaggio per la descrizione del progetto, il file scritto nello standard Crospen, è il fulcro effettivo del sistema di generazione automatico, poiché sarà l'input da dare in pasto al programma, nonché la documentazione di progetto, fondamentale per l'approccio simil-goal-oriented adottato con CROSS.
L'applicazione sarà eseguita su un server **node**, e genererà a sua volta due componenti per l'applicazione: una componente server (con annessa gestione del database) in Express e una componente per il client implementata con framework VUE.
L'idea dunque è quella di generare una applicazione web REST facendo uso di una applicazione web REST.
Dunque, l'output dell'applicazione sarà effettivamente un'applicazione pronta ad essere eseguita, completamente funzionante e con la sola presentazione grafica eventualmente da personalizzare (compito estremamente semplice grazie all'utilizzo di VUE come framework di presentazione dell'applicazione).
L'applicazione avrà quattro moduli fondamentali, che verranno spiegati in paragrafi specifici a loro dedicati: **Crosser** (il parser per il file Crospen), **DBG** (il generatore di database), **Crossback** (il generatore della parte backend dell'applicazione) e **CrossVue** (il generatore della parte frontend dell'applicazione).

5.1 Crosser

Il parser **Crosser** si occupa di leggere il file in input per poi restituire le strutture dati e gli oggetti JSON necessari alla generazione delle interfacce e delle funzioni di backend. Il file dato in pasto all'applicazione deve essere scritto in YAML e rispettare la sintassi di Crospen Api. Dovrà inoltre fornire tutte le informazioni necessarie alla generazione dell'applicazione.

Nello specifico, verranno utilizzate le tre sezioni del documento in questo modo:

- **Servers**: fornisce i vari entry-point dell'applicazione. L'indirizzo dunque verrà utilizzato per fornire al nostro backend Express l'indirizzo dei server del database e della applicazione.

- **Paths**: descrive le operazioni CRUD associate agli url del backend, ognuno dei quali utilizzato per dare input al server per ottenere delle risposte dal database fornendo anche le informazioni sulla persistenza dell'operazione compiuta.

- **X-navigation**: che descrive quello che sarà il routing front-end dell'applicazione, le varie pagine e con quali componenti grafiche realizzarle, utilizzando il concetto di vista, che come promesso andremo ad analizzare a breve.

Il parser dunque ha il compito di creare tre oggetti JSON, contenenti ognuno le informazioni necessarie ai tre generatori per poter operare alla generazione effettiva dell'applicazione.
Ci si avvale, vista la compatibilità con il formato OpenApi, dello strumento di validazione e di parsing SwaggerParser [Smab] che fornisce uno strumento di validazione e di parsing per il formato OpenApi. La validazione ha il fondamentale ruolo di interrompere subito il processo di creazione ritornando un codice di errore in caso il file passato in input sia mal scritto o presenti errori, mentre il parser per OpenApi fornito, rende più semplice dividere il file e ritornare degli oggetti JSON e delle strutture di dati .
Le strutture dati generate nello specifico sono: un dizionario in cui la chiave è il ruolo dell'utente e con valore le sue capabilities (nei confronti delle operazioni sul server e sul database), un grafo rappresentante la struttura di navigazione del sito, una stringa (o un array di stringhe) contenente il valore dell'entry point. È ancora da comprendere come salvare gli oggetti e gli attributi su cui il database dovrà andare ad operare, così come inizializzare le tabelle sulle basi di dati.
La caratteristica dei file Crospen Api rispetto ai dati con cui interagiscono le operazioni , come già esposto precedentemente, è data dalla presenza dei riferimenti $ref che permettono di associare degli oggetti descritti poi nel tag components specifico all'interno del documento stesso. Tramite una funzione fornita da [Smab] possiamo risolvere tutti i riferimenti (è fornita anche una guardia in caso di riferimenti circolari che restituisce un

errore). Quindi verrà generato un oggetto JSON contenente le operazioni e i dati su cui operano dereferenziati e un secondo oggetto JSON contenente i nostri oggetti.

5.1.1 Crossback

La generazione del server avviene in maniera piuttosto semplice. Sarà implementato con framework Express, scelta dovuta al largo uso e alla prestanza di questo framework per i servizi backend REST.
Infatti ci permette di gestire in automatico le richieste HTTP, e questo rende ben più semplice la scrittura del codice di generazione.
Crossback prende in input il file JSON delle operazioni, contenente i path, e genera un file chiamato "app.js" all'interno della cartella Backend. Per creare cartelle e file si farà ricorso alla libreria "fs" di node. Il file avrà intestazione standard contenente l'inserimento dei moduli necessari al funzionamento dell'applicazione: **cors**(necessario nel momento in cui delle API facciano riferimento a più server), **json-parser** (che si occupa della lettura e della gestione degli oggetti JSON), **nodemon** (che ricompila il server automaticamente ad ogni modifica, utile per la fase di test una volta generata l'app e per verificare le eventuali modifiche), **fs**(libreria di gestione del file system virtuale utilizzato sul web) ed **Express**(Framework per la gestione delle richieste HTTP).
Una volta scritta l'intestazione standard all'interno del file, si passa dunque alla generazione del middleware Express, che gestirà dunque le varie richieste al server. Una prima funzione standard sempre uguale per tutte le applicazioni generate con CSG sarà la richiesta di login: prenderà il path dato come entry point e gestirà la comunicazione con il database per la verifica nome utente - password per poi garantire il login all'utente a cui verrà assegnato un Token, che permette al database di accomunare a lui le capabilities e all'interfaccia di restituire le viste a lui dedicate (in caso di user interface differenziata a seconda del ruolo).
Dopodiché vengono presi i path all'interno dell'oggetto in input e viene creata dunque una serie di funzioni con sintassi standard Express. Ad esempio, nel caso di una operazione **get** avremo una sintassi tipo: *app.get("url", function(req, res)//funzione/i di interazione con il client e richieste al database)*
All'interno delle funzioni ci saranno le varie richieste da comunicare al database (dette

query) sviluppato con MongoDB data l'ampia fetta di mercato e di utilizzo che ha, e invia al client un file JSON con dentro l'elenco degli elementi da stampare a video (in caso si tratti di una operazione get) e come stamparli, mentre si occuperà di salvare sul database gli aggiornamenti sui dati in caso di eliminazione, modifica o upload di un nuovo dato.

5.1.2 DBG

Come già accennato, il database generato è basato su tecnologia MongoDB.
DBG si occupa dunque di due importantissime operazioni: inizializzare le tabelle, e creare le funzioni per gestire le richieste.
In primo luogo, verranno create le tabelle degli utenti (una per ogni ruolo) per il login, che conterranno le password cifrate, curando di creare delle funzioni di cifratura standard. Dopo di che, verranno create le tabelle contenenti gli oggetti definiti nello schema con tutti i campi necessari. Ad esempio, facendo riferimento all'esempio del negozio di musica, potremmo avere il caso degli articoli in vendita: per ogni articolo avremo un nome, una data di produzione, un costo, un flag per verificare la presenza in magazzino, il numero di articoli presenti in negozio ecc... Tutte queste informazioni, elaborate da Crosser, sono contenute nel file JSON passato al modulo DBG, che dovrà semplicemente occuparsi di elaborare a sua volta l'oggetto e lanciare i comandi MongoDb per la generazione delle tabelle. Per evitare incresciosi errori, nella tabella degli utenti viene automaticamente generato uno user con tutti i permessi di cui password e nome utente verranno restituiti allo user che sta generando il sito così da permettere la piena gestione del database e i test di funzionamento manuali. In caso si voglia scegliere di utilizzare un provider di terze parti per mettere in hosting il servizio di database, basterà cambiare nel codice il link del server a cui fa riferimento il database stesso e inizializzare manualmente le tabelle utilizzando l'interfaccia del servizio scelto. A quel punto si potrà comunque fare riferimento alle funzioni per le query generate nel file "**database.js**", facendo tuttavia attenzione all'utilizzo di MongoDB come linguaggio per comunicare con il database.
Il file "database.js" sarà incluso nel server, e si occupa per l'appunto di definire tutte le funzioni per la comunicazione con il database. La scelta di separare il codice del middleware del server da quello delle funzioni di interazione con il database serve a conferire

al codice maggiore leggibilità e maggiore modularità, cercando dunque di aumentare la qualità generale del codice, che in applicazioni generate automaticamente spesso viene a mancare.
Ogni funzione dunque (che verrà chiamata dal middleware Express) si occuperà semplicemente di formulare le richieste per interagire con il database stesso.

5.2 CrossVue

Arriviamo ora alla sezione più interessante del progetto: la generazione del client.
Per iniziare, si ricorda al lettore che per ogni applicazione da generare richiedente come fase iniziale il login, l'interfaccia di questa operazione sarà standard e uguale per ogni sito generato con CSG. Tuttavia, come il resto del codice generato dal tool, anche questa è completamente modificabile sia in termini grafici, sia in termini di funzionalità, lasciando al cliente la scelta di implementare eventuali login tramite socialnetwork, OTP (One Time Password) o simili.
Il client è basato sul framework "**VUE**.js". La scelta di questo framework deriva dalla sua semplicità e dalla possibilità di riuso dei template.
L'obbiettivo è quello di creare una single page web application, dove data una schermata statica di navigazione a menù, vengono caricate a schermo le interfacce per visualizzare le informazioni o meglio, per dirla rispettando il modello CROSS, i concetti. Il client avrà una libreria di template grafici che chiameremo fragments ispirandoci al concetto di vista parziale in Android.
Bisogna immaginare dunque un assemblaggio delle viste come se queste fossero dei mattoncini colorati di plastica, dove ogni colore corrisponde ad un modello di vista CROSS differente, e il nostro generatore il bambino che non fa altro che incastrarli tra di loro, seguendo le disposizioni di "colori" a lui fornite. Queste disposizioni sono ritrovabili all'interno degli schemas di CrospenApi.
In questo paragrafo quindi andiamo prima a comprendere il funzionamento del framework scelto, per poi vedere i fragments vari rispettando le specifiche di visualizzazione CROSS, per poi comprendere come questi verranno richiamati a schermo dall'applicazione.

Sarà inoltre usata la libreria Bootstrap[1] per una maggiore estetica di base, visto e considerato che questa ha integrazione nativa con il framework.

5.2.1 VUE.JS

Iniziamo qui dunque una breve panoramica sul framework, comprendendo i suoi punti di forza e il suo funzionamento. Tutte le informazioni relative al framework sono accessibili da [vue20], che invito il lettore a consultare per una più completa comprensione dell'argomento trattato.
VUE è un framework progressivo per la generazione di interfacce utente, e al contrario di altri framework monolitici, è adatto all'integrazione in ambienti preesistenti.
Il funzionamento di VUE si basa sul caricamento degli oggetti nel **DOM**[2] usando sintassi da **template**(dei documenti precompilati in cui variano dei valori, caricati a tempo di esecuzione a seconda delle esigenze). VUE si occupa in maniera approfondita di creare collegamenti tra oggetti passati e oggetti del **DOM**, rendendo l'applicazione completamente reattiva.
Un documento VUE è strutturato in tre sezioni:

1. La sezione **template**, che definisce l'interfaccia grafica, comportandosi esattamente come precedentemente descritto.

2. La sezione **script**, in cui inserire la logica della sezione (codice javascript), ponendo al suo interno ciò che la pagina deve restituire, eventuali funzioni di reazione ad eventi ecc..

3. La sezione **style**, in cui specificare il comportamento grafico in css[3] del template

VUE fornisce dei costrutti che permettono di generare viste identiche in sequenza con contenuto diverso, come ad esempio **v-while** o **v-for**, che hanno lo stesso funzionamento

[1] ¡https://getbootstrap.com/docs/4.4/getting-started/introduction/¿ È un semplice framework per la realizzazione di siti responsive e mobile first

[2] secondo [W3C09] il modello ad oggetti del documento (appunto il DOM) è lo standard ufficiale del W3C (World Wide Web Consortium) di rappresentazione dei documenti web come modello orientato agli oggetti

[3] Cascade Style Sheet, linguaggio di markup adottato nell'ambito del web e più generalmente nella descrizione di interfacce per descrivere il comportamento grafico di un oggetto da mostrare a schermo

dei costrutti while e for in qualsiasi linguaggio imperativo come Java o C++. Inoltre abbiamo la possibilità di fare dei caricamenti di oggetti condizionali tramite l'uso della formula **v-if**, anch'esso dal funzionamento analogo ai linguaggi precedentemente citati.
Il costrutto **v-on** è un altro forte costrutto di VUE, che semplifica l'interazione tra utente e pagina, e fa in modo di mandare in esecuzione la funzione descritta nel momento in cui si verifica l'evento descritto(che può essere un click su un bottone, oppure un inserimento testuale, o anche un posizionamento statico del cursore in una determinata zona della schermata). VUE si occupa della completa manipolazione del DOM, quindi non ci si deve preoccupare di gestire la modifica effettiva di questo.
Un altra funzionalità fornita da VUE degna di nota è "**v-model**" che semplifica il binding tra un oggetto di input ed il DOM. Il costrutto è definito da [vue20] come zucchero sintattico per aggiornare i dati all'inserimento di dati da parte dell'utente.
Tuttavia la caratteristica più importante di VUE, almeno per quanto riguarda la sua integrazione nella applicazione web, sta nell'utilizzo dei **components**. Un **component** è una istanza di VUE con delle opzioni predefinite. Tramite l'utilizzo del parametro "**props**", possiamo caricare dinamicamente il contenuto di un **component**.

5.2.2 Fragments in CrossVue

Tramite l'uso dei components Vue, possiamo implementare quelli che chiamiamo Fragments nel modello CROSS: ovvero, la visualizzazione di una o più istanze del concetto in maniera comprensibile. Abbiamo diversi tipi di vista, e per ognuna di questi vi è un **component** VUE di riferimento da richiamare nel momento in cui il server restituisca il dato da mostrare, con le relative informazioni di visualizzazione del dato stesso. Andiamo dunque finalmente ad analizzare le componenti di vista in CROSS, e per ognuna di esse, vediamo l'implementazione in VUE.js che andranno a comporre i nostri **components**. L'operazione di vista, e la restituzione dei modelli di vista nell'output in CROSS condividono i modelli scelti. Mentre nella prima, viene identificato il modello relativo al concetto con cui si vuole operare, o che si deve ritornare in seguito ad una operazione, nell'output abbiamo una più chiara modellazione della interfaccia vera e propria. Una volta dunque analizzati i Fragments vedremo come questi vanno a comporre l'interfaccia.

Individuale completa

È la visualizzazione completa di un concetto. Sono visualizzate tutte le proprietà e le informazioni ad esse relative. CROSS stesso consiglia l'utilizzo di progressive disclosure in caso le informazioni fossero in alto numero. Inoltre, se permesso all'utente, deve permettere un update globale, ovvero di tutte le proprietà del concetto.
In CrossVue, il fragment Individuale Completo, è un template semplice, realizzato in una sezione di schermo dove viene individuato all'interno di una tabella, a sinistra il nome del concetto e a destra il suo valore. In caso di un ampio numero di elementi, verificabili con un semplice controllo sul file JSON passato in input a CrossVue, viene comunicato l'utilizzo delle progressive disclosure (come consigliato dal modello CROSS stesso), implementate tramite l'elemento "Accordion" di Bootstrap. L'intestazione sarà

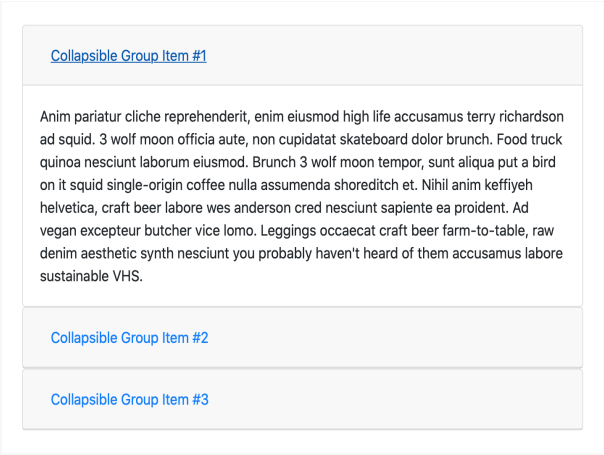


Figura 5.1: *Disclosure realizzata con Collapse preso dalla documentazione ufficiale di Bootstrap: http://www.getbootstrap.com*

la parola chiave del concetto, e al suo interno sarà presente una tabella contenente tutti i concetti appartenenti al concetto superiore. Inoltre, viene fornita la funzione di aggiornare l'oggetto se possibile all'interno di questa vista. Inoltre, se l'utente è autorizzato, le caselle contenenti i valori dei concetti saranno cliccabili e modificabili.
La scelta qui è ricaduta su un semplice bottone "update", che alla sua pressione avvia l'operazione di aggiornamento sul database e modifica i campi modificati nell'oggetto. Ogni casella scrivibile sarà del tipo Lookup, che analizzeremo a breve.

Individuale ridotta

È una visualizzazione parziale di un concetto. Sono visualizzate solo le proprietà e le informazioni indicate come fondamentali. Da una individuale ridotta deve essere possibile passare ad una completa, e deve inoltre essere possibile passare ad una operazione di update globale o locale (ovvero solo sui campi mostrati).
In questo caso, nel fragment realizzato non vi è progressive disclosure, ma viene visualizzato il concetto come una riga di una tabella, dove andiamo ad inserire per ogni colonna il valore relativo. Questi saranno modificabili, come per la completa, e sarà possibile salvare l'aggiornamento delle variabili grazie ad un bottone "update". Inoltre cliccando sul campo non modificabile a sinistra che individua il concetto, sarà possibile passare ad una individuale completa.
Anche qui è presente l'utilizzo di Lookup nelle zone scrivibili dall'utente.

Lista

È una visualizzazione di più concetti. È un elenco di concetti rappresentati da visualizzazioni individuali ridotte. Deve garantire il passaggio ad una visualizzazione individuale, fornire operazioni sulla lista stessa come ordinamenti, raggruppamenti e creazione di una istanza.
Il fragment lista sarà una semplice tabella con all'interno i fragment individuali ridotti. Per le righe avremmo il concetto macro, mentre sulle colonne saranno individuabili i valori dei componenti dei concetti. È ovvio che per visualizzare una lista, è fondamentale che i concetti siano dello stesso gruppo e abbiano gli stessi campi di valore.
Le operazioni sono fornite da appositi bottoni presenti subito sopra la tabella, che permettono di ordinare (ad esempio in ordine alfabetico) i risultati, di filtrare per valore di un campo ecc...

Expense	Amount	Category	Comment	Date	RIMBORSO
pannelli fonoassorbenti	$26.00	strumentazione studio upgrade amazon	https://www.amazon.it/12-pezzi-acustica-assorbimento-acustico-pannelli/dp/B078VTHLCB	dic 05, 2019	[redacted]
Masterizzatore cd	$19.99	strumentazione studio	https://www.amazon.it/Ark-Cartoncino-stampa-spesso-formato/dp/B07R91Y65M/ref=sr_1_1?__mk_it_IT=ÅMÁŽÕÑ&crid=1MOQ0Q0WPQNGY&keywords=cartoncini+400g&qid=1575578301&sprefix=cartoncini+400%2Caps%2C159&sr=8-1	dic 05, 2019	[redacted]
Colla	$6.20	materiale marketing amazon		dic 05, 2019	[redacted]
Adesivi	$18.00	materiale marketing		dic 05, 2019	[redacted]
Stampa Copie Fisiche	$22.00	materiale marketing		dic 06, 2019	[redacted]
Materiale Bricoland	$16.00	materiale studio		dic 14, 2019	[redacted]
Stickers Alaska	$18.00	marketing Entertainment		dic 20, 2019	[redacted]
+ New					
	SUM $324.19				

Figura 5.2: *Esempio di lista nel caso di un rimborso spese.*

Lookup

È una visualizzazione di concetti da selezionare e utilizzare in seguito. Utilizza una visualizzazione individuale ridotta (anche di un solo campo come ad esempio il nome).

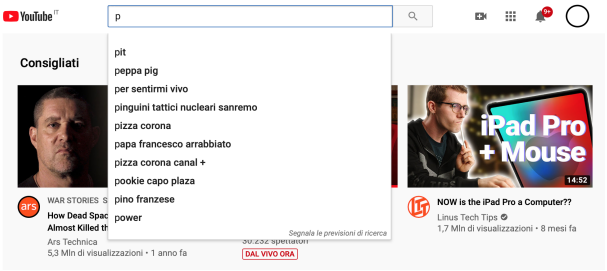


Figura 5.3: *Lookup in ricerca dei video presenti sulla piattaforma Youtube*

L'implementazione in questo caso è il semplice menu a tendina che appare nel momento in cui un utente inizi a scrivere qualcosa. Viene redatta una ricerca tra i concetti del sistema, e vengono restituiti i suggerimenti relativi al campo che si sta modificando. Un

esempio di lookup appartenente al mondo reale è la ricerca di video su youtube.com

Recap

Sono raggruppamenti fatti sulle istanze del concetto. Possono essere dei conteggi, delle sommatorie, ultime creazioni... Un esempio è individuabile nella figura 5.2 con la sommatoria del totale di spesa a fondo tabella.

Capitolo 6

Conclusioni, Valutazioni e Sviluppi Futuri

In questa tesi sono stati analizzati il processo di sviluppo CROSS e il formalismo Crospen Api, ed è stato presentato il programma di generazione automatica di applicazioni web REST Cross Standard Generator.
Dopo aver analizzato le precedenti soluzioni al task della creazione di interfacce utente, abbiamo analizzato CROSS, osservando come la fase di analisi dei requisiti, svolta nel modello proposto, porti ad un processo in cui l'utente viene messo al centro. Si è analizzato ogni componente del suo acronimo in modo panoramico, al fine di fornire una comprensione generale del modello di produzione che lo strumento CSG adotta.
Proseguendo, si è visto il formalismo Open Api, lo standard per la documentazione di API REST, osservando il concetto di API, REST, e dando una panoramica sugli strumenti di cui è provvisto. Questo è stato fatto al fine di comprendere a pieno il funzionamento di Crospen, il linguaggio che CSG adotta nella scrittura del file di documentazione richiesto in input per la generazione della applicazione, che è stato descritto subito dopo.
Per finire è stato esposto il funzionamento di CSG, analizzando le scelte delle tecnologie utilizzate per quest'ultimo.

Il lavoro svolto in questa tesi è stato per lo più teorico e di progettazione. Per motivi di tempo il prototipo realizzato durante le fasi di sviluppo, risulta essere troppo

grezzo, e non essendo soddisfatto del codice da me prodotto, ho ritenuto fosse meglio posticipare l'effettivo sviluppo del prototipo di CSG al periodo post-laurea, con l'obbiettivo di riuscire ad implementare il tool in tutto e per tutto, rilasciando così anche il sorgente del codice.

L'obbiettivo che che mi sono prefissato è quello di riuscire a formare una comunità Open Source attorno al progetto, così da riuscire a far entrare CROSS come metodologia di sviluppo, Crospen Api come metodologia di documentazione e CSG come processo iniziale nello sviluppo di software web all'interno delle realtà aziendali.

CSG comporterebbe, in caso venisse adottato, un gran numero di benefici al mondo dello sviluppo di applicazioni web REST, a partire dal rigore di applicazione del modello CROSS, fino alla standardizzazione nella realizzazione di tali applicazioni, aumentando il grado di leggibilità del codice. Questo porterebbe ad un approccio maggiormente user-centered nello sviluppo di questo tipo di piattaforme, contribuendo ad ampliare il mondo dell'usabilità.

Inoltre, la diretta conseguenza della standardizzazione effettiva nella scrittura delle API, comporterebbe un più facile "passaggio di mano" del codice tra programmatori che collaborano direttamente alla sua scrittura, evitando problemi quali la mancanza di commenti all'interno delle sezioni di codice, la scarsa leggibilità dovuta al personale modo di scrivere di ogni programmatore... In più, con un processo standard di generazione ed un modello standard di scrittura, vengono meno anche gli errori basilari riguardanti l'usabilità del prodotto, e la sua funzionalità in senso stretto.

Per quanto riguarda l'interfaccia della applicazione, risulta ovvio che questa avrà come scopo il costituire uno scheletro dell'interfaccia finale. Lo scopo dell'interfaccia generata infatti è quello di costituire una garanzia di usabilità del prodotto, non una sua presentazione grafica gradevole. Per raggiungere questo requisito è necessario infatti l'intervento di un grafico umano. È sicuramente in discussione (e tra le proposte di implementazioni future) lo sviluppo di uno strumento di design grafico drag and drop (a mo di Java-FX

di cui abbiamo parlato precedentemente) per la modifica della presentazione grafica del sito web, fornendo strumenti di design e collocamento icone, modifiche di colore, modifiche di posizionamento degli elementi sulle interfacce ecc... Data la natura Open Source dell'intero progetto, questo strumento guadagnerebbe eventualmente una sua comunità di sviluppo, che potrebbe portare poi alla creazione di temi preconfezionati e altre funzionalità interessanti presenti in questo tipo di strumenti.

Bibliography

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[Krz04] Daniel S. Weld Krzysztof Gajos. *SUPPLE: Automatically Generating User Interfaces*. 2004. URL: http://www.cs.tufts.edu/comp/250VA/papers/supple-gajos.pdf.

[W3C09] W3C. *About DOM*. 2009. URL: https://www.w3.org/DOM/.

[WL10] Benjamin Weyers e Wolfram Luther. "Formal modeling and reconfiguration of user interfaces". In: *2010 XXIX International Conference of the Chilean Computer Science Society*. IEEE. 2010, pp. 236–245.

[Mau+17] Gioacchino Mauro et al. "Extending a user interface prototyping tool with automatic MISRA C code generation". In: *arXiv preprint arXiv:1701.08468* (2017).

[vue20] vue. *Introduction to VUE.JS*. 2020. URL: https://vuejs.org/v2/guide/.

[OAI] OAI. *OpenApi 3.0.0 Example*. URL: https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/api-with-examples.yaml.

[Smaa] SmartBear. *OpenApi specification*. URL: https://swagger.io/docs/specification/about/.

[Smab] SmartBear. *Swagger Parser*. URL: https://www.npmjs.com/package/swagger-parser.

Ringraziamenti

Questa tesi vuole essere un punto di inizio ad un percorso di ricerca e passione nel mondo dell'informatica, che in questi anni di studio sono riuscito a far entrare sempre più tra i miei interessi. Vuole essere il proposito di continuare ad interessarmi al mondo dello sviluppo e della ricerca di soluzioni a problematiche e, più di tutto, vuole essere un monito per me stesso, atto a ricordarmi sempre che si possono cambiare le vite delle persone, anche creando soluzioni semplici, o anche compiendo un solo piccolo gesto verso qualcuno.

Spendo dunque due parole per ringraziare chi ha cambiato la mia di vita, in questi tre anni di percorso all'università di Bologna.

Inizio con il ringraziare i miei genitori e mia sorella Federica, perché senza di loro non avrei intrapreso questo percorso.

Ringrazio Eleonora, Marco, Denis, Guglielmo, Lorenzo e Giuseppe per aver reso i momenti difficili all'interno della facoltà più leggeri ed affrontabili.

Ringrazio Andrea, Giacomo e Gianluca per aver sempre trovato il tempo di condividere con me il loro di tempo.

Ringrazio Erik, Marina, Flavia, Umberto, Giuseppe e Federica, per aver condiviso con me la quotidianità, e per avermi reso (ognuno a suo modo) la persona che sono oggi.

Ringrazio Davide, Andrea, Nicola, Loris e Marta, per darmi ogni giorno un motivo nuovo di proseguire nella mia passione laterale all'informatica e all'università.

Ringrazio il professor Fabio Vitali, per avermi dato fiducia, e per avermi spinto ad affrontare questa sfida, che spero davvero di continuare nel periodo successivo alla laurea.

Ringrazio Lorenza per essere stata qui con me quando ne ho avuto più bisogno, augurandomi che rimanga al mio fianco ancora un po'.

Ringrazio infine tutti coloro che sono stati di passaggio nella mia vita.

So long, and thanks for all the fish