Università Politecnica delle Marche

Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Componenti software per la sincronizzazione di database integrati in applicazioni mobile con un database centralizzato

Software components for the synchronization of databases embedded into mobile applications with a centralized database

Relatore:
Prof. Alessandro Cucchiarelli

Candidato:
Alessandro Contu

Anno accademico 2019/2020

Indice

Abstract

La presente tesi descrive il lavoro svolto durante il periodo di tirocinio curriculare presso Infoservice S.r.l, partner del gruppo Zucchetti e azienda leader nei software gestionali e sistemi ERP per aziende, che sviluppa soluzioni integrate gestionali e Web per una vasta gamma di settori e funzioni aziendali (controllo di gestione, business intelligence, logistica di magazzino, gestione della produzione e portali Web).

Il lavoro svolto prevede la creazione di un modulo funzionale per un'applicazione Web che permetta la sincronizzazione con un database esterno, implicando quindi anche lo sviluppo di una logica lato server che permetta di attuare la funzione richiesta.

La realizzazione di queste funzionalità fa uso di diverse tecnologie, come i framework Ionic e React Native per la creazione dell'interfaccia grafica dell'applicazione, linguaggi di programmazione Web (HTML, CSS e JavaScript), linguaggi SQL e il Java per lo sviluppo di servlet.

Si illustreranno quindi le varie pagine che vanno a comporre l'applicazione, ciascuna che permette di eseguire azioni diverse, ma tutte incentrate ad interfacciarsi con il database interno all'app stessa o alla sua sincronizzazione con la base di dati esterna. Successivamente verranno mostrate le logiche di sincronizzazione lato server, ovvero quegli algoritmi che svolgono la funzione di accettare le richieste del client (l'applicazione) ed eseguire delle elaborazioni che portano ad una corretta risposta, formattata in JSON.

Capitolo 1

Introduzione

Negli ultimi anni è sempre più diffuso ed incentivato il fenomeno della digitalizzazione, ovvero il processo di conversione in un formato digitale di oggetti e processi. Per un'azienda ciò non significa solo una conversione di documenti cartacei in file elettronici, ma una completa revisione delle strategie e dei processi produttivi. Per favorire questa trasformazione, sono state sviluppate numerose tecnologie, che permettono di raggiungere risultati in maniera più efficace ed efficiente. In questo contesto trovano spazio gli ERP, ovvero dei sistemi informativi integrati che rendono possibile il dialogo, per mezzo di un unico strumento, tra tutti i processi aziendali.

Per soddisfare le esigenze di determinati lavori effettuati in mobilità, si è reso inoltre necessario sviluppare nuove applicazioni che potessero essere utilizzate tramite dispositivi mobile, come cellulari e tablet. Questi applicativi devono poter essere utilizzabili sia in presenza che in assenza di connessione e devono quindi avere un sistema che permetta di sincronizzare, anche successivamente al loro inserimento, le nuove informazioni con un sistema centralizzato (che può essere per l'appunto un ERP).

Si diffonde dunque l'uso di sistemi software distribuiti, ovvero costituiti da un insieme di processi che dialogano tra loro. Spesso questi assumono dei modelli centralizzati formati da un grande sistema centrale e sistemi distribuiti che si rapportano con esso e lo aggiornano, incorrendo perciò in problemi legati alla replica e consistenza dei dati, oltre al controllo delle concorrenze.

In questo contesto opera la **Infoservice S.r.l.** azienda italiana che si occupa di sviluppare soluzioni integrate gestionali e Web, in settori specifici quali la manutenzione antincendio

e il food & beverage, oltre ad offrire un servizio completo dalla consulenza alla produzione di software personalizzati.

Tra i suoi applicativi può vantare due software: **Taylor** e **Mywo**, che rappresentano bene il tipo di applicazioni sopra descritte. Il primo viene usato principalmente per la gestione delle manutenzioni antincendio e si relaziona nativamente al gestionale *Ad Hoc Revolution* sviluppato da **Zucchetti,** azienda Italiana leader di settore; il secondo è una soluzione mobile per la gestione degli ordini da parte di un agente. Entrambe hanno una caratteristica fondamentale per questa tipologia di applicativi per i quali è necessario il funzionamento in mobilità: consentono il loro utilizzo anche offline e i dati raccolti, come per esempio degli ordini da evadere, saranno elaborati successivamente in presenza di rete.

Il presente progetto è incentrato sullo sviluppo di un modulo software per una nuova applicazione, creata nella sua totalità, che permetta alla stessa di interfacciarsi con il proprio database interno e sincronizzarlo con una base di dati esterna. L'applicazione dovrà quindi essere in grado di gestire le possibili iterazioni che l'utente potrà avere: esso potrà creare un database interno all'app ed inizializzarlo, ad esempio tramite una prima sincronizzazione con il database esterno e quindi creando una copia speculare dello stesso. L'utente potrà effettuare varie modifiche al database interno, tramite le classiche operazioni di modifica, cancellazione ed inserimento, oltre a poter visualizzare graficamente i dati contenuti nelle tabelle. Infine sarà disponibile un'interfaccia che permetterà di sincronizzare le modifiche apportate alla base di dati interna con il database esterno.

Per rendere possibile quanto appena descritto però, non basterà sviluppare l'applicazione e creare una base di dati esterna, ma sarà necessario configurare un server che sia in grado di accettare le richieste del client (ovvero l'applicazione) e gestirle

Saranno utilizzate perciò numerose tecnologie che serviranno per implementare tutto ciò: i due framework che renderanno possibile la creazione dell'applicazione e creeranno la sua interfaccia grafica *Ionic* e *React* (quest'ultimo sviluppato da *Facebook*), i linguaggi

Web come *HTML, CSS e TypeScript* (che estende il JavaScript), *l'SQL* per l'interfacciamento ai database, la creazione di un server *Tomcat* e l'utilizzo di *Java* per sviluppare delle *JavaServer Page*.

Da quanto sopra esposto si evince che lo scopo del lavoro è la creazione nella loro totalità delle logiche e tecnologie di sincronizzazione, tra una Web app e un database esterno per mezzo di un serve

Capitolo 2

Contesto applicativo

2.1 Sistemi informativi integrati ERP

I sistemi informativi integrati **ERP** (**Enterprise Resource Planning**) rappresentano soluzioni applicative che le organizzazioni utilizzano per gestire le attività commerciali e integrare su base aziendale i processi operativi ed amministrativi che regolano lo svolgersi delle attività aziendali. Questi sistemi si basano su un'unica struttura di dati definita che condivide, in genere, un database comune e centralizzato riconducendo quindi in un unico schema logico i flussi di informazione che hanno origine nelle diverse aree funzionali.

In italiano possiamo tradurre letteralmente ERP come "pianificazione delle risorse dell'impresa", ma questi sistemi non devono essere considerati come semplici software gestionali. Infatti mentre questi ultimi tendono soprattutto a raccogliere e aggregare informazioni nelle diverse aree di gestione, gli ERP sono capaci di integrare i diversi sottoinsiemi informativi e flussi organizzativi dell'azienda. Quindi le differenze principali sono in termini di struttura e di capacità di gestione dei processi: i gestionali, nonostante siano in grado di gestire più di un'area aziendale, non possono farlo in maniera integrata e soprattutto non possono offrire una gestione globale dei processi.

Un problema che viene risolto dai sistemi informativi integrati è quello della gestione dei dati: i software gestionali tradizionali rischiano di avere una duplicazione dei dati, dovuta ad una gestione non integrata dei differenti processi. Questo aumenta esponenzialmente la possibilità di avere problemi di coerenza ed errori. Gli ERP invece, gestendo i processi

in maniera integrata e archiviando i dati inseriti in un unico database, garantiscono l'univocità del dato. [1]

Un altro fattore che caratterizza i sistemi ERP è la struttura modulare, visibile in *Figura 1*, che li rende flessibili e altamente personalizzabili in base alle esigenze di ogni area aziendale. Infatti a ciascuna di esse corrisponde un modulo o un'applicazione e questo consente all'impresa di attuare una strategia *best of breed* [1], garantendo sia una possibile copertura incrementale delle aree aziendali che una crescita nel tempo con l'implementazione di moduli aggiuntivi che soddisfino le nuove esigenze.

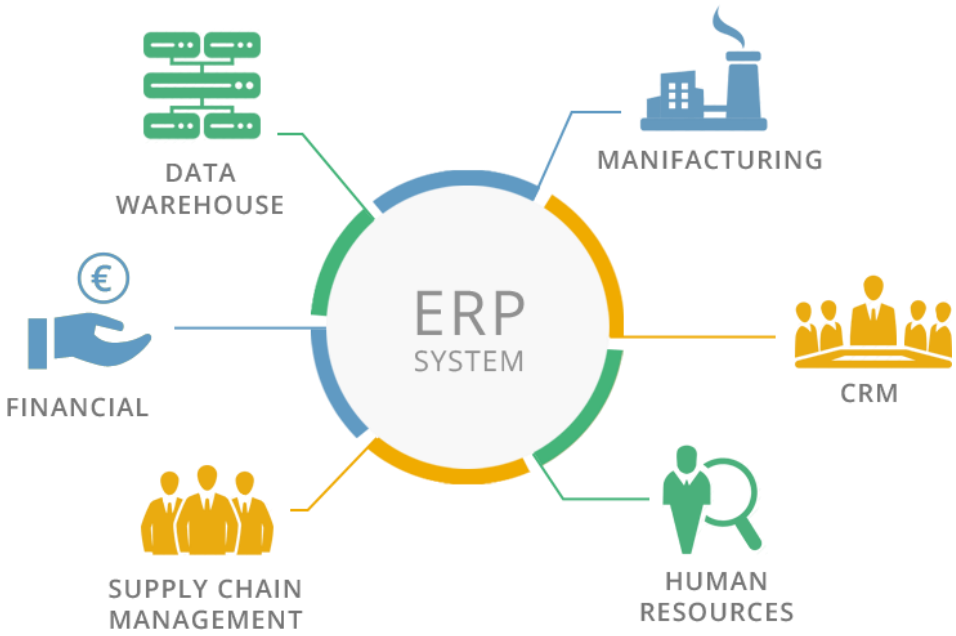

Figura 1: Ambiti applicativi di un sistema ERP

Infine grazie al collegamento con interfacce Web based e grazie alla logica del cloud, gli ERP possono dialogare in tempo reale con soggetti esterni all'azienda come fornitori, rivenditori e agenti.

[1] Con best of breed si intende qualsiasi prodotto che viene considerato il migliore nella sua categoria

2.2 Applicazioni mobile

Quando si parla di applicazioni mobile, comunemente chiamate app, ci si riferisce ad una determinata tipologia di applicazione software che è disegnata per essere eseguita su un dispositivo mobile, come uno smartphone o un tablet.

Questo tipo di applicazioni devono essere realizzate in maniera più leggera in termini di risorse hardware utilizzate rispetto alle applicazioni per desktop computer, ma permettono, ovviamente, un utilizzo in mobilità che è indispensabile in determinati settori lavorativi. Si pensi ad esempio all'agente di commercio, la cui attività consiste nel promuovere determinati prodotti e servizi e stabilire dei contratti commerciali tra i clienti e l'azienda: ha bisogno di strumenti utilizzabili in mobilità per visualizzare le informazioni relative ai clienti, accedere ai cataloghi, generare preventivi e ordini. È quindi necessario un software che sia creato ad hoc per le loro necessità e che sia fruibile da mobile. Inoltre le App permettono di collegarsi con i diversi sistemi aziendali come ERP e *CRM*[2], dando quindi la possibilità di avere una gestione completa del lavoro a prescindere dalla presenza in azienda o dall'uso di un computer.

A livello tecnico, possiamo suddividere le App mobile in tre tipologie:

- **App Native**: rappresentano quelle applicazioni che sono sviluppate appositamente per un certo sistema operativo, principalmente Android o iOS, e sarà quindi garantito l'accesso diretto a tutte le funzionalità del dispositivo, oltre ad avere prestazioni ottimali.
- **Web App**: queste non sono un vero e proprio programma come le precedenti, ma sono un collegamento verso un applicativo remoto che non è quindi fisicamente installato sul dispositivo, ma è accessibile tramite un browser. Il vantaggio di un Web App consiste nel fatto di non dipendere dalla capacità di calcolo del dispositivo, in quanto il nucleo elaborativo dell'applicazione è presente su server remoti

[2] Il Customer Relationship Management rappresenta una strategia di business che si avvale di strumenti informatici allo scopo di creare relazioni durature con i clienti analizzando le informazioni che li riguardano

- **App Ibride**: sono una sorta di via di mezzo delle precedenti, in quanto sfruttano sia componenti nativi che tecnologie Web essendo scritte nei linguaggi CSS, HTML e JavaScript. Le App ibride risultano quindi delle Web App integrate in App native

Quest'ultima tipologia risulta essere particolarmente interessante in quanto, nonostante le performances delle App ibride risultino inferiori rispetto alle App native, hanno comunque pieno accesso alle funzioni del dispositivo in uso e sono fortemente adattabili ai diversi tipi di piattaforme.

2.3 TAYLOR e MYWO

Taylor e MYWO sono due software prodotti dall'azienda Infoservice: il primo è un applicativo dedicato alla gestione delle manutenzioni antincendio, il secondo è un sistema di gestione ordini per agenti e clienti.

2.3.1 Taylor

Come sopra accennato Taylor è un software gestionale per le aziende di manutenzione antincendio e si compone di un'App e di un gestionale. Sviluppato tutto in tecnologia Zucchetti, Taylor consente ai manutentori di avere una gestione ottimizzata dei dati permettendo interventi rapidi ed efficaci, avendo a disposizione gli strumenti e i dati di cui hanno bisogno: la digitalizzazione fornisce infatti una maggior sicurezza e riduce al massimo le possibilità di errore, mettendo a disposizione uno strumento in grado di archiviare, confrontare ed elaborare i dati in automatico.

Attraverso l'utilizzo di tablet con sistemi di lettura barcode, si ha una facile gestione delle non conformità e la creazione di una reportistica ad hoc per le varie esigenze, come schematizzato nella *Figura 2*. Inoltre grazie all'integrazione tra software ed ERP, che viene aggiornato in tempo reale sulle attività svolte, si possono ottimizzare i processi ed incrementare gli interventi.

Figura 2: Architettura di Taylor

Tra le varie funzionalità offerte dall'applicativo troviamo:

- La gestione automatica delle scadenze, con possibilità di filtrare in base al cliente, tecnico, articolo, ubicazione o periodicità
- La visualizzazione in agenda degli interventi, assegnati al singolo tecnico o al gruppo di lavoro, con la geolocalizzazione su mappa degli interventi per facilitare la programmazione degli stessi
- Un accesso real-time ai dati contabili e del database assistenze del cliente da parte del manutentore
- La possibilità di utilizzare l'applicazione offline, con una successiva sincronizzazione dei dati quando la rete tornerà nuovamente disponibile

Grazie ad applicazioni come questa, i manutentori non scrivono più su fogli di carta ma compilano i dati direttamente su tablet e questa maggiore digitalizzazione porta ad una maggiore sicurezza ed efficienza. Inoltre l'adozione di un metodo di lavoro in cloud in quei settori che prevedono uno spostamento frequente dei dipendenti, come nel caso della manutenzione, costituisce una forte spinta alla crescita aziendale: risulta infatti che le aziende aumentano del 12% in fatturato rispetto a quelle che non usano queste tecnologie, e si registra anche un incremento del 9% dei posti di lavoro. [2]

2.3.2 MYWO

Mywo rientra in quel campo di applicazioni aziendali che permette di effettuare ordini ai clienti direttamente o attraverso gli agenti. La gestione degli ordini si rivela un'operazione notevolmente più efficiente quando gli interventi manuali sono ridotti al minimo; grazie alla maggiore automazione infatti, non c'è la necessità di un ufficio che gestisca i preventivi singolarmente per ogni cliente, dando informazioni su prezzi, proprietà del prodotto o scontistiche. Inoltre grazie alle soluzioni software, l'agente avrà sempre a portata di mano tutte le informazioni richieste dal cliente e necessarie per portare a termine l'ordine correttamente.

Mywo è un'applicazione multipiattaforma utilizzabile da tablet, smartphone e pc inoltre, essendo costruito come Web App, risulta altamente *scalabile*[3]. Tra le funzionalità offerte dal software troviamo:

- **La gestione degli ordini**. È l'elemento principale del sistema. Prevede la gestione completa e diversificata degli ordini (vedi *Figura 3*), con funzionalità di ricerca, portafoglio ordini, tracciamento e tracking dello stato, oltre che la registrazione dei dati storici del cliente. Tramite la maschera dell'elenco ordini è possibile fare ricerche avanzate e la creazione di un ordine segue una procedura guidata per facilitare la compilazione

[3] Un software risulta scalabile quando è facilmente aggiornabile a bassi costi e velocemente

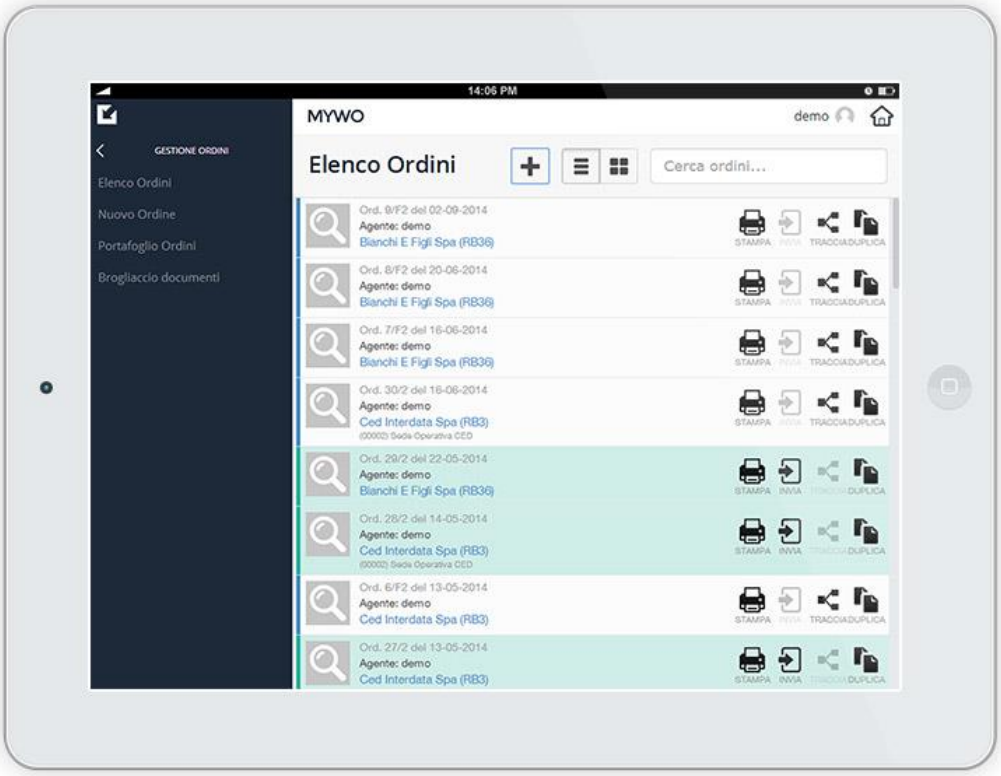

Figura 3: Pagina gestione ordini visualizzata da tablet

- **La gestione di offerte e preventivi**. È possibile creare o sincronizzare documenti di offerta e preventivi, gestendone i *le versioni*[4] successive e mantenendo traccia delle variazioni nel tempo
- **Il carrello e il tracking**. È implementata una gestione avanzata del carello ordini di un cliente, accompagnato da un tracciamento in tempo reale dello stato dell'ordine integrato con il sistema gestionale aziendale
- **Una agenda personale**. Ogni utente ha a disposizione un'agenda con le proprie attività e i propri ordini
- **Utilizzo di documenti e allegati**. La gestione documentale consente di abbinare agli articoli delle schede tecniche , note, immagini e tutto ciò che risulti utile a finalità tecniche e commerciali

[4] Ogni volta che un documento viene modificato, la versione corrente viene salvata e ne viene generata una nuova che diventa la versione attuale; si può avere la possibilità di controllare anche le versioni precedenti del documento

- **Estratto conto**. È possibile generare in tempo reale accurati report sullo stato di avanzamento dei pagamenti, sul cliente, su bolle e fatture, ed effettuare analisi statistiche aggregare sui vari dati
- **Geolocalizzazione e navigazione**. È presente un'integrazione automatica delle sedi dei clienti con Google Maps, con la possibilità di filtrare sulla mappa e di impostare la navigazione presso un cliente o visualizzare distanze e tempi di percorrenza
- **Utilizzo online e offline**. L'applicazione funziona perfettamente anche in assenza di connessione, infatti dà comunque la possibilità di creare un ordine che sarà poi elaborato successivamente in presenza di rete

L'applicativo va in contro a quelle che sono le necessità di *dematerializzazione*[5] della documentazione, digitalizzando l'intera procedura di acquisizione ed elaborazione degli ordini. Questo porta ad una maggiore efficienza sia nel vendere che nell'acquistare, oltre che ad un notevole risparmio: si rende praticamente nullo l'utilizzo di carta e inchiostro, oltre che delle ore lavorative di chi si deve occupare di stampanti e archivi, aumentando di conseguenza anche la produttività individuale e aziendale.

Mywo risulta nativamente integrato con gli ERP Ah Hoc Revolution e Ad Hoc Enterprise sviluppati dalla Zucchetti.

[5] Sostituzione dei tradizionali supporti cartacei per la registrazione di atti e documenti, con file digitali

Capitolo 3

Obiettivi del progetto

3.1 Introduzione

Sia Taylor che Mywo sono delle applicazioni che permettono una gestione completa di ogni necessità che un manutentore nel primo caso, o un agente di commercio nel secondo, si trova ad affrontare. La varietà di funzioni che questi software mettono a disposizione, rendono la loro realizzazione particolarmente complessa; per questo motivo lo sviluppo del progetto a cui la relazione fa riferimento, tratta una sola di queste funzionalità.

È da precisare che questo non significa che lo sviluppo si è incentrato unicamente sulla creazione del modulo software che permette la sincronizzazione tra l'applicazione e il database esterno, ma sono state create anche funzioni accessorie, oltre all'attenzione prestata per quanto riguarda la parte di *UI* ed *UX* [6].

La problematica affrontata ha portato alla creazione di tutte quelle componenti software necessarie per la connessione e l'aggiornamento tra un'applicazione Android e un database esterno; questo ha richiesto lo sviluppo delle seguenti componenti:

- Un'**applicazione Android**, con la quale l'utente interagisce
- Un **database interno all'applicazione**, che per mezzo di un'interfaccia grafica possa essere inizializzato, eliminato o aggiornato
- Un **database esterno**, che conterrà le stesse tabelle presenti nel database interno all'applicazione ma con la possibilità di avere tuple differenti in numero e contenuto

[6] La User Interface rappresenta la parte visiva e grafica dell'applicazione, con la quale l'utente si approccia; con User eXperience si definiscono quei processi volti a migliorare la facilità di navigazione ed utilizzo dell'app per l'utente

- Un **Web server** e un **servlet**, in grado di gestire le richieste provenienti dall'applicazione e interfacciarsi con il database esterno

3.2 Applicazione Android

Per quanto riguarda l'applicazione essa presenta, nel momento del suo avvio, un *carousel*[7] che permette all'utente di capire qual è la struttura interna dell'applicazione e quali funzioni potranno essere attivate. Dopo che l'utente avrà visualizzato interamente questo percorso introduttivo, sarà possibile utilizzare effettivamente l'applicazione. Questa risulta articolata in tre pagine differenti, alle quali è possibile accedere tramite i tre tabs posizionati nella parte bassa dello schermo, come schematizzato in *Figura 4*; si può notare che il tab corrispondente alla pagina che si sta visualizzando viene colorato di blu.

La prima pagina è quella che l'utente si troverà di fronte ad ogni avvio dell'applicazione, ed è dedicata all'inizializzazione del database interno. È possibile creare la base di dati attraverso due metodologie differenti:

- Si riempiono le tabelle con dei dati già salvati nell'applicazione, che sono stati pre-settati in fase di sviluppo; questa opzione ha più un senso didattico che una vera utilità pratica, ma potrebbe essere necessaria nel momento in cui si utilizza per la prima volta l'applicazione e si vogliono inserire dei dati nelle tabelle, ma non si ha la possibilità di inizializzare il database sincronizzandolo con il database esterno
- Si instaura una connessione con il database esterno e, per ciascuna delle tabelle indicate dall'App, vengono restituiti in formato JSON tutti i dati contenuti nelle stesse tabelle della base di dati esterna all'App, che verranno poi inseriti

[7] Un carousel è un insieme di slides che vengono visualizzate in sequenza

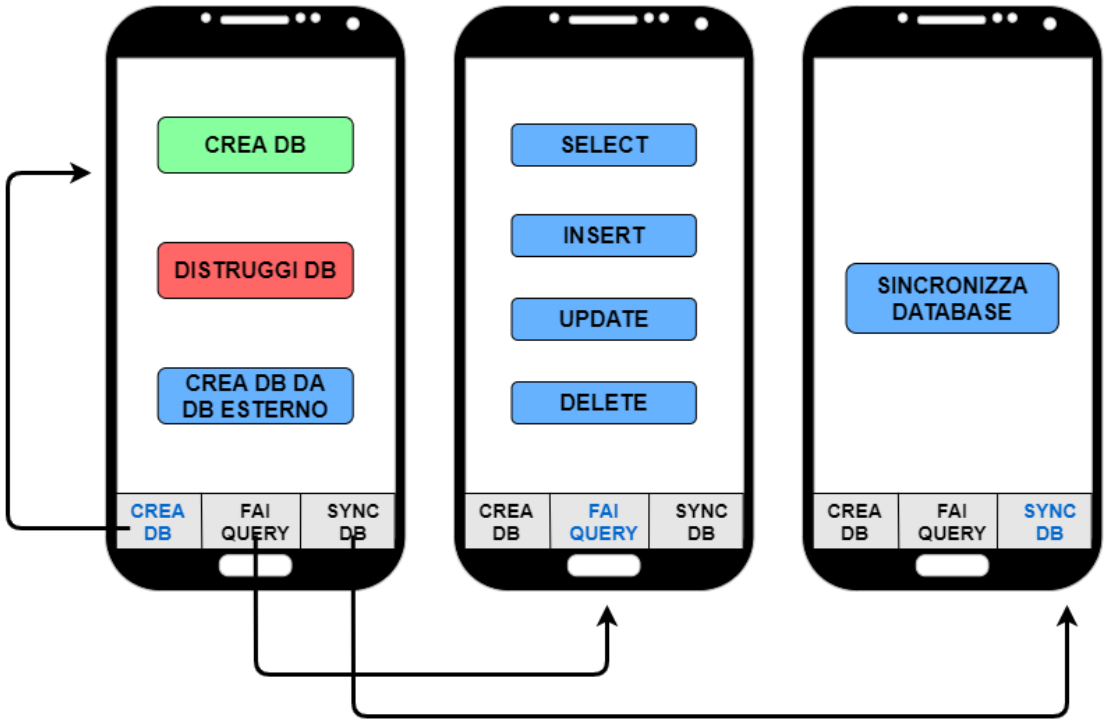

Figura 4: Schema di navigazione interna tramite tab

Oltre a queste due funzionalità, nella prima pagina è possibile anche eliminare l'intero database interno facendo il drop di tutte le tabelle. Eseguire questa operazione è un'azione ovviamente molto pericolosa, in ottica di un'applicazione realmente utilizzabile, ma risulta utile in fase di test per vedere se ogni sistema che dialoga con il database interno lo fa correttamente e, in caso di necessità, è possibile azzerare il database per poterlo ricreare.

La seconda pagina, accessibile dal tab centrale, permette di interfacciarsi al database interno all'App e di modificarlo. Ognuno dei bottoni presenti su questa vista porta ad un'altra pagina, che varia in base al tipo di operazione da effettuare. In tutti i casi è possibile selezionare la tabella dove eseguire la query e, in base alla scelta fatta, verranno visualizzati in modo dinamico gli attributi della tabella con i relativi campi di input. A seconda dei campi di input inseriti, della tabella selezionata e dell'operazione scelta, verrà costruita una query che verrà poi eseguita.

L'ultima pagina che è possibile visualizzare risulta più minimale graficamente, ma permette di eseguire la funzione principale dell'applicazione: la sincronizzazione tra App e database esterno. La logica che sta dietro quest'operazione è articolata nelle varie componenti, interne ed esterne all'App. Per quanto riguarda la parte mobile, le procedure si susseguono in questo modo per ciascuna tabella presente nel database interno all'App:

- Si estraggono le chiavi primarie della tabella
- Si estraggono tutte le tuple che sono state inserite, aggiornate o eliminate dall'ultima sincronizzazione in poi
- Per ciascuna tupla viene creato dinamicamente un URL contenente i dati formattati nel modo *"nomeAttributo=valoreAttributo"* insieme ai valori delle chiavi primarie e alla data e ora dell'ultima sincronizzazione. Con l'URL così formato l'App fa una richiesta HTTP al server, utilizzando il metodo GET
- In base alla risposta, che ritorna in formato JSON, vengono fatte delle operazioni di inserimento, aggiornamento o eliminazione sulla base di dati interna.

3.3 Database interno ed esterno

Per lo sviluppo dell'Applicazione si è deciso di creare un database che avesse degli attributi simili a quelli che potrebbe avere una base di dati per la gestione degli ordini, ma con una struttura tabellare semplificata, implementando però tutte le caratteristiche necessarie per un testing adeguato.

Per quanto riguarda il database interno all'App, questo sarà composto da tre tabelle (visibili in *Figura 5*), la prima *"Master"* rappresenta le caratteristiche generali degli ordini, con i seguenti attributi:

- **Order_id**, che è rappresentato tramite un intero ed è la chiave primaria della tabella, sarà quindi diverso per ogni tupla

- **Client** e **Client_id**, il primo è il nome del cliente relativo all'ordine mentre il secondo identifica il cliente tramite un codice id e vengono rappresentati tramite stringa (*varchar*) e numero intero rispettivamente
- **Order_date**, è la data in cui è stato effettuato ordine e viene rappresentato tramite il tipo predefinito "*date*"

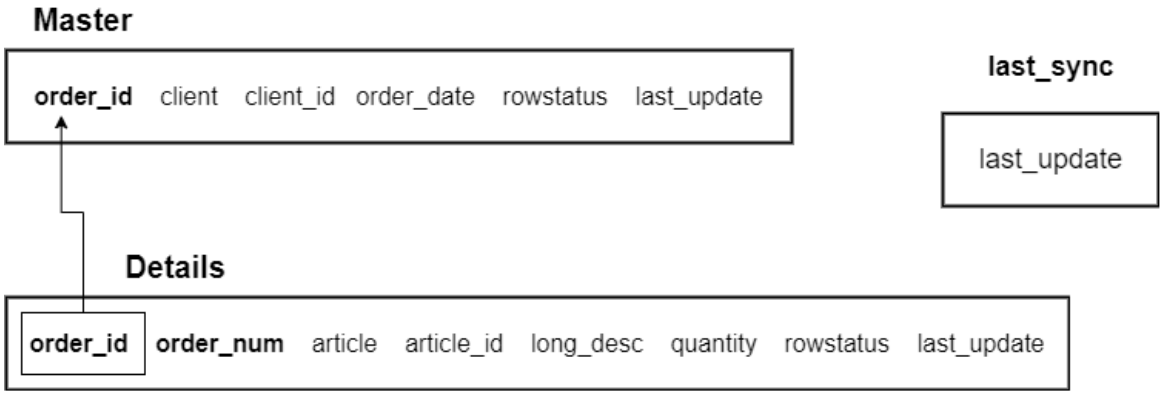

Figura 5: Schema tabellare con indicazioni chiavi primarie ed esterne

La seconda tabella *"Details"* invece elenca, per ciascun ordine, gli elementi che lo compongono tramite gli attributi:

- **Order_id** e **Order_num**, che insieme compongono la chiave primaria della tabella, con order_id che è una chiave esterna relativa all'omonimo attributo della tabella Master. Il primo indica l'id dell'ordine mentre il secondo un ulteriore numero identificativo interno all'ordine, entrambi rappresentati da interi
- **Article** e **Article_id**, identificano l'articolo di un ordine e il suo codice identificativo, rappresentati da una stringa e un numero intero
- **Long_desc** è una descrizione dell'articolo e viene rappresentata tramite una stringa che può essere composta da massimo 200 caratteri
- **Quantity** è la quantità di un determinato articolo presente nell'ordine e per questo si utilizza un numero intero

Infine la terza tabella *"last_sync"* ha un unico campo "*last_update*" che conterrà un'unica tupla, che rappresenta la data e l'ora dell'ultima sincronizzazione tra il database dell'App e il database esterno.

Così facendo, nonostante il database risulti formato da sole tre tabelle, vengono utilizzati vari tipi di dato ed elementi dei DBMS relazionali come chiavi primarie e chiavi esterne. È da specificare inoltre che nelle tabelle *master* e *details* sono presenti due campi, che non sono mostrati all'utente finale, ma vengono utilizzati dall'applicazione per la sincronizzazione dei database. I due attributi infatti danno delle informazioni riguardo la tupla e indicano se essa deve essere considerata tra le tuple da confrontare oppure no:

- **Last_update** indica la data e l'ora di quando la tupla è stata aggiornata oppure inserita, utilizzando il tipo *timestamp*
- **Rowstatus** dà un'informazione su qual è stata l'ultima operazione relativa ad una tupla, quindi se è stata inserita, aggiornata o eliminata; questo dato verrà rappresentato tramite un semplice char *"i" , "u" , "d"*

Per il database esterno verranno create tre tabelle, due di esse con stessi nomi e struttura delle tabelle *master* e *details* presenti nella base di dati interna dell'applicazione, quindi con stessi attributi, tipi e vincoli di integrità; inoltre ad entrambe le tabelle verrà aggiunto un campo "*dirty_bit"*, necessario per le logiche di sincronizzazione. Solo queste due dovranno essere utilizzate per la sincronizzazione, mentre la terza tabella presente nel database esterno non dovrà essere modificata e servirà quindi per testare il corretto funzionamento del sistema di sincronizzazione. Inoltre in questo database dovranno essere inseriti dei *trigger* [8] che, nel momento in cui una tupla verrà inserita, aggiornata o eliminata, setteranno il valore di *'rowstatus'* e *'last_update'* della tupla stessa.

[8] Un trigger è una proceduta che viene eseguita in maniera automatica al verificarsi di un determinato evento

3.4 Web server

Il collegamento e la comunicazione tra l'applicazione e il database esterno non possono avvenire in maniera diretta, ma c'è bisogno di un elemento terzo che si interponga tra e faccia dialogare questi due elementi. Questo scopo viene raggiunto da un Web server, ovvero quell'applicativo software che viene eseguito su un server e gestisce richieste HTTP da parte di un client, che nel caso di questo progetto è rappresentato dal dispositivo mobile, e restituisce una risposta di tipo JSON come rappresentato in *Figura 6*.

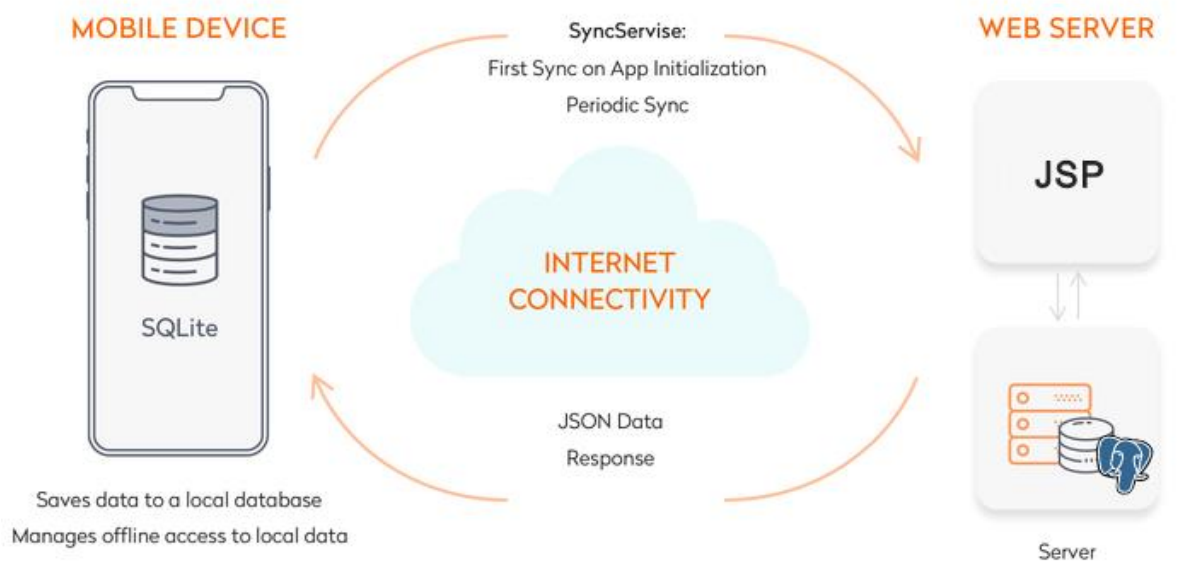

Figura 6: Schema collegamento dispositivo mobile e Web server

All'intero del server Web dovranno operare dei servlet, ovvero degli oggetti scritti in linguaggio Java che permettono di eseguire delle elaborazioni lato server.

Questi elementi dovranno gestire le richieste di sincronizzazione da parte dell'applicazione nel modo seguente:

- Acquisiscono le informazioni della tupla da sincronizzare dalla richiesta GET
- Si collegano con il database esterno ed estraggono delle tuple da una determinata tabella

- Confrontano la tupla da sincronizzare, proveniente dal database dell'App, con le tuple estratte
- In base al risultato del confronto, modificheranno il database esterno oppure restituiranno una risposta JSON che indicherà all'App in che modo aggiornare la propria base di dati

Alla fine di questo processo, ripetuto per tutte le tabelle necessarie, risulteranno sincronizzati ed aggiornati entrambi i database.

Capitolo 4

Strumenti utilizzati

4.1 Ionic React

4.1.1 Introduzione

Come descritto nei capitoli precedenti, il progetto prevede la creazione di un'applicazione Android. Tra le tante opzioni possibili per svilupparla, si è scelto di utilizzare il *framework* [9] Ionic integrato con il framework React Native.

Ionic si incentra sul lato front-end, quindi gestisce la parte di UX e UI dell'applicazione, utilizzando tecnologie Web come HTML, CSS e JavaScript. Inoltre tramite questo framework si sviluppano applicazioni ibride, con tutti i vantaggi che questo comporta, come già spiegato nel *capitolo 2.2* .

React Native è anch'esso un framework utile per lo sviluppo di applicazioni mobile ed è tra i framework integrati con Ionic, oltre ad Angular e Vue. React utilizza Typescript, un linguaggio che estende il JavaScript permettendo di definire i tipi di dato, oltre ad usare classi e moduli. [3]

4.1.2 Ionic CLI

Per semplificare lo sviluppo dell'applicazione nelle sue diverse componenti, è stata sviluppata la **Ionic CLI**, acronimo di Command Line Interface, ovvero un'interfaccia a riga di comando che facilita la costruzione dell'App.

[9] Un *framework* è un'infrastruttura intesa come insieme complesso di librerie e programmi di supporto, che consentono di sviluppare ed integrare diverse componenti software evitando allo sviluppatore finale di doverle creare da zero

Per poter installare Ionic CLI è però necessario utilizzare un package manager, ovvero un insieme di strumenti che automatizzano il processo di installazione e configurazione dei software. Si utilizza **npm**, abbreviazione di Node Package Manager, che è il gestore di pacchetti predefinito per **Node.js**, ovvero l'ambiente di runtime JavaScript costruito sul motore Javascript V8 di Chrome. [4]

Quindi dopo aver scaricato ed installato Node.js sulla macchina di sviluppo del progetto, sarà possibile installare Ionic CLI tramite il comando:

```
$ npm install -g @ionic/cli
```

Successivamente, si crea lo scheletro di un'App tramite il comando:

```
$ ionic start myApp tabs --type=react
```

Questo creerà la base dell'applicazione chiamata "myApp", che non sarà vuota in quanto abbiamo selezionato "tabs" e quindi sarà formata da tre pagine tra le quali è possibile navigare grazie ai tre tab posizionati in fondo alla pagina. Inoltre, come spiegato precedentemente, Ionic permette l'integrazione con vari framework e poiché quello di default è Angular, è necessario impostare React tramite il parametro "--type".

Infine per mandare in esecuzione un Web server che consentirà di utilizzare l'applicazione, è possibile utilizzare il seguente comando, dopo essersi posizionati nella cartella contenente il progetto:

```
$ ionic serve
```

Tramite questo comando si avvierà un server in *localhost*[10] all'indirizzo **http://localhost:8100** che verrà lanciato direttamente sul browser e consentirà l'utilizzo

[10] *Localhost* è il nome di dominio che viene usato dalle applicazioni per comunicare con un Web Server installato ed attivo sullo stesso sistema su cui sono in esecuzione.

dell'applicazione; inoltre in base alle modifiche fatte sui file di progetto, queste verranno automaticamente caricate e aggiorneranno l'applicazione in tempo reale. [5]

4.1.3 UI Components

Le applicazioni sviluppate con Ionic sono costituite da blocchi predefiniti di alto livello chiamati **componenti**, che permettono di costruire in modo rapido un'interfaccia utente per l'App. Ciascun componente è formato da uno o più **custom elements**: lo standard dei Web component dà la possibilità di creare elementi che incapsulano le funzionalità volute nella pagina HTML, senza dover creare un insieme intricato e nidificato di elementi. [6] Ciascun custom element permette di utilizzare metodi, eventi e proprietà CSS già definiti, oppure di crearne di nuovi. Ionic rende disponibili un insieme completo di componenti che permettono la creazione di un'App in ogni suo aspetto.

4.1.4 Typescript e JSX

Tra le tecnologie introdotte da React vi è JSX, termine che sta per JavaSctipt eXtension. Questa è un'estensione sintattica di Javascript che permette di creare dei componenti utilizzando una sintassi simile all'HTML; è infatti possibile creare delle variabili che sono valorizzate con una sintassi simile all'XML, anche annidando i tag, alle quali viene assegnato un riferimento ad un oggetto JavaScript che rappresenta un elemento React. I *React elements* sono una rappresentazione virtuale dei nodi del DOM reale.

Nel progetto verranno sviluppati dei file con estensione "*tsx"* che danno la possibilità di utilizzare sia la sintassi del JavaScript che quella del Typescript, poiché quest'ultimo verrà comunque compilato in JavaScript prima di essere interpretato.

4.2 Database interno ed esterno

4.2.1 Introduzione

Poiché la base di dati è uno degli elementi centrali di tutto il lavoro, risulta necessaria un'analisi più dettagliata, soprattutto alla luce del fatto che i due database da sincronizzare sono di tipologie diverse: SQLite per l'applicazione e PostgreSQL come database esterno. Entrambi sono RDBMS, termine che indica un relational database management system, ovvero un sistema di gestione di basi di dati basato sul modello relazionale. Questo tipo di modello logico di strutturazione dei dati, si fonda sui due concetti di relazione e tabella e permette di distinguere il livello fisico dei dati, che li organizza in opportune strutture fisiche, e il livello logico, al quale l'utente fa riferimento. [7]

Questi tipi di DBMS richiedono la definizione di strutture, come tabelle, per poter contenere i dati e manipolarli; con le tabelle ogni colonna può rappresentare un tipo di dato diverso e ogni tupla può essere identificata univocamente grazie ad uno o più attributi che prendono il nome di chiavi. Inoltre entrambi i database che vengono utilizzati nello sviluppo di questo progetto, godono dei requisiti ACID per le transazioni:

- Atomicità, che garantisce l'indivisibilità delle operazioni racchiuse in una transazione
- Consistenza, che garantisce che la transazione lasci il database in uno stato integro e non corrotto
- Isolamento, che limita la visibilità delle modifiche tra transazioni concorrenti
- Durabilità, che garantisce la persistenza delle modifiche di una transazione andata a buon fine

Nonostante questi aspetti comuni, ogni RDBMS può variare per le differenti implementazioni dei tipi di dato, oltre a vari altri aspetti che rendono ogni sistema di gestione migliore in casi diversi. Una prima semplice differenza tra i database utilizzati nello sviluppo si nota già nel momento di creazione delle tabelle. Gli stessi attributi, che hanno quindi gli stessi tipi di dato, risultano inizializzati in modo differente: per esempio per inserire la data e l'ora attuali, nel caso di SQLite si userà la notazione

'strftime('%H:%M:%S %Y/%m/%d','now')' , mentre con PostgreSQL bisognerà scrivere *'now()::timestamp'*.

4.2.2 SQLite

SQLite è una libreria software compatta scritta in linguaggio C, con caratteristiche peculiari per essere usata nei sistemi embedded, tanto da essere il database engine più usato al mondo[8]. Tra i suoi punti di forza troviamo:

- Velocità di esecuzione
- Estrema portabilità, in quanto l'intero database risulta costituito da un unico file su disco
- Supporta database che possono essere anche molto grandi, fino a 140 TB
- È multipiattaforma, infatti viene usato in ogni tipologia di sistema mobile o desktop

Risulta quindi una scelta ottimale per tutte le applicazioni che richiedono portabilità, come nel caso di quella sviluppata, e uno dei suoi punti deboli in questo caso non risulta un vero e proprio svantaggio: non si ha nessuna gestione degli utenti multipli ma, data la finalità di questa tipologia di software, questa funzionalità non è necessaria.
SQLite risulta ben integrato con Ionic React: è infatti possibile installarlo all'interno della nostra App tramite i seguenti comandi:

```
$ npm install cordova-sqlite-storage
$ npm install @ionic-native/sqlite
$ npm install @ionic-native/core
$ ionic cap sync
```

Sarà poi possibile creare, accedere e modificare il database tramite delle funzioni predefinite, che saranno utilizzabili e richiamabili all'interno delle pagine dell'applicazione. I metodi che verranno utilizzati sono:

- **SQLite.create()** che permette di creare o accedere ad un determinato database
- **SQLite.executeSql()** che viene usato per le query alla base di dati

Da notare però che per potersi relazionare con il database Ionic ha bisogno di **Capacitor**, ovvero un runtime di App multipiattaforma, che permette di creare Web App eseguite in modo nativo su iOS ed Android, supportando quindi un accesso completo agli *SDK* [11] nativi di ciascuna piattaforma. Non è quindi possibile sviluppare l'applicazione utilizzando un server locale, ma è necessario utilizzare o emulare un dispositivo sul quale eseguire l'App.

4.2.3 PostgreSQL

Per quanto riguarda il database esterno, la scelta è ricaduta su PostgreSQL. Quest'ultimo è un RDBMS avanzato ed orientato agli oggetti e si basa su una tecnologia capace di gestire in modo efficiente compiti multipli: riesce ad ottenere la concorrenza tra utenti senza lock di lettura grazie alla realizzazione dell'*MVCC* [12]. Si rivela quindi un'ottima scelta per questa implementazione, in quanto ci troviamo nel caso in cui ci sono molteplici dispositivi mobile, ciascuno con il proprio database locale, che si devono collegare allo stesso database esterno.

Verranno utilizzati anche due strumenti che hanno uno scopo simile, ma due differenti implementazioni. **Psql** è il terminale interattivo di PostgreSQL e viene installato direttamente insieme ai package del database; ha quindi un'interfaccia a riga di comando

[11] Gli *SDK*, ovvero i Software Development Kit, rappresentano un insieme di strumenti per lo sviluppo software e possiedono un compilatore che permette di tradurre il codice sorgente in codice eseguibile

[12] Il *Controllo della Concorrenza MultiVersione (MVCC)* è un metodo di controllo della concorrenza comunemente usato dai sistemi di gestione per fornire un accesso concorrente alle basi di dati e fa in modo tale che ogni istruzione SQL vede un'istantanea dei dati (una versione del database) com'era poco tempo prima, indipendentemente dallo stato corrente dei dati sottostanti. Ciò impedisce alle istruzioni di visualizzare dati incoerenti prodotti da transazioni simultanee che eseguono aggiornamenti sulle stesse righe di dati, fornendo l'isolamento delle transazioni per ogni sessione di database

e permette di scrivere query in modo interattivo, oltre a fornire una serie di meta-comandi per facilitare la codifica di script e l'automazione di un'ampia varietà di attività. [9]

PgAdmin è un'applicazione open source, scritta in C++, che al contrario di Psql ha un'interfaccia grafica e consente di amministrare in modo semplificato il database. È possibile creare un database da zero, creare le tabelle ed eseguire operazioni di ottimizzazioni su di esse.

4.3 Android Studio

Come descritto alla fine del *capitolo 3.2.2*, per poter sfruttare le potenzialità di Capacitor è necessario installare l'applicazione su un dispositivo Android. Questo passaggio può essere realizzato in due modi:

- Utilizzando un vero dispositivo mobile, che verrà collegato alla macchina di sviluppo tramite cavo USB
- Utilizzando un dispositivo mobile emulato tramite software

Nel corso dello sviluppo si è scelto di seguire questa seconda strada ed è stato scaricato il programma **Android Studio:** è un *IDE* [13] che permette di creare applicazioni Android native e può creare devices Android virtuali, necessari per il nostro sviluppo.

Dopo aver scaricato questo software, prima di effettuare di deploy dell'App sul dispositivo Android emulato, il progetto deve essere configurato attraverso i seguenti comandi:

```
$ npm install -g @ionic/cli native-run cordova-res
$ ionic capacitor add android
$ ionic capacitor copy android
```

[13] Un *IDE* , Integrated Development Environment, è un ambiente di sviluppo integrato ovvero un software che supporta il programmatore in fase di sviluppo di un'applicazione.

Infine è stato scelto come dispositivo da emulare un **Pixel 2** con Android nella sua versione 8.0 (Oreo).

Per mandare in esecuzione l'applicazione sul dispositivo appena creato, bisogna eseguire il comando:

```
$ ionic capacitor run android
```

Sarà necessario utilizzare questo comando ogni volta che si apportano modifiche al codice dell'applicazione, in quanto verrà fatta nuovamente la build, copia e deploy dell'App Ionic sul simulatore Android.

Una volta che l'App è in esecuzione sul dispositivo, è possibile fare il *debugging*[14] attraverso il **Chrome DevTools**. Il browser Chrome ha infatti uno strumento che supporta lo sviluppo di applicazioni Android che vengono eseguite su device esterni o emulati: andando sull'indirizzo ***"chrome://inspect"*** mentre l'App è in esecuzione, è possibile visualizzare eventuali messaggi di errore o messaggi dati dai **console.log()** inseriti all'interno del codice del progetto.

4.4 Server Web

4.4.1 Introduzione

Un Web server è un'applicazione software che gestisce le richieste di trasferimento di contenuti Web da parte di un client e questa comunicazione avviene attraverso il protocollo HTTP. L'*HyperText Transfer Protocol* è il protocollo principale per la trasmissione di informazioni in un'architettura client-server Web: quando il client invia una richiesta, specifica un *metodo* HTTP, che indica al server che tipo di azione vuole che sia effettuata. Tra i vari metodi esistenti, quello che verrà utilizzato per l'applicazione

[14] Il *debugging* è l'attività che consiste nell'individuazione e correzione di uno o più errori (detti bug) che possono essere rilevati all'interno del codice.

è il metodo **GET**, che consente di accodare all'indirizzo della pagina (*URL)* i diversi parametri contenenti i dati che si vogliono trasmettere.

4.4.2 Tomcat

Apache Tomcat è un server Web open source sviluppato dalla Apache Software Foundation e risulta essere principalmente un contenitore di tecnologie scritte in Java come servlet e JSP, ma non implementa completamente la specifica *Java EE* [15].

L'architettura di Tomcat consiste in una serie di componenti funzionali, come descritto in *Figura 7*, che possono essere combinati insieme seguendo delle regole definite. La struttura di ogni server viene definita nel file "*server.xml"* , posizionato all'interno della cartella "*/conf".*

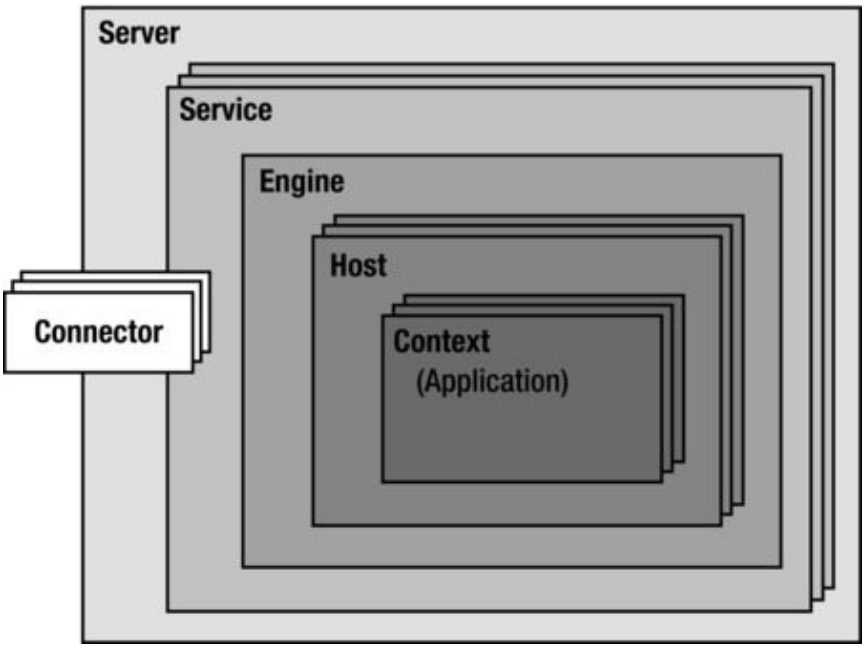

Figura 7: Architettura di Tomcat

Dopo aver scaricato ed installato Tomcat, la sua corretta esecuzione sarà verificabile accedendo all'URL **http://localhost:8080**. Inoltre per un corretto funzionamento tra

[15] La *Java Platform Enterprise Edition* è un insieme di specifiche le cui implementazioni, sviluppate in Java, sono utilizzate nella programmazione Web

applicazione e Web server, è necessario andare ad inserire all'interno del file "*Web.xml",* anch'esso posizionato nella cartella "*/conf"*, le seguenti righe di codice:

```
<filter>
    <filter-name>CorsFilter</filter-name>
    <filter-class>org.apache.catalina.filters.CorsFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>CorsFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Il **CORS**, termine che indica il Cross-Origin Resource Sharing, è un meccanismo che usa header HTTP addizionali, per indicare al browser che un'applicazione Web in esecuzione su un dominio dispone dell'autorizzazione per accedere alle risorse selezionate da un server di origine diversa. Quindi un'applicazione invia una cross-origin HTTP request quando richiede una risorsa che ha un'origine differente dalla propria. [10] Tramite il codice precedente, è possibile abilitare il CORS su Tomcat.

4.4.3 Servlet

Un servlet è un oggetto scritto in linguaggio Java che opera all'interno di un Web server, migliorando ed estendendone le funzionalità, permettendo di creare contenuti dinamici per pagine Web. [11]

I servlet vengono eseguiti all'interno della Java Virtual Machine sul server, come mostrato in *Figura 8*, quindi risultano essere sicuri e portatili; infatti questi oggetti operano solo all'interno del dominio del server e, al contrario delle applets, non necessitano che il Web browser supporti Java.

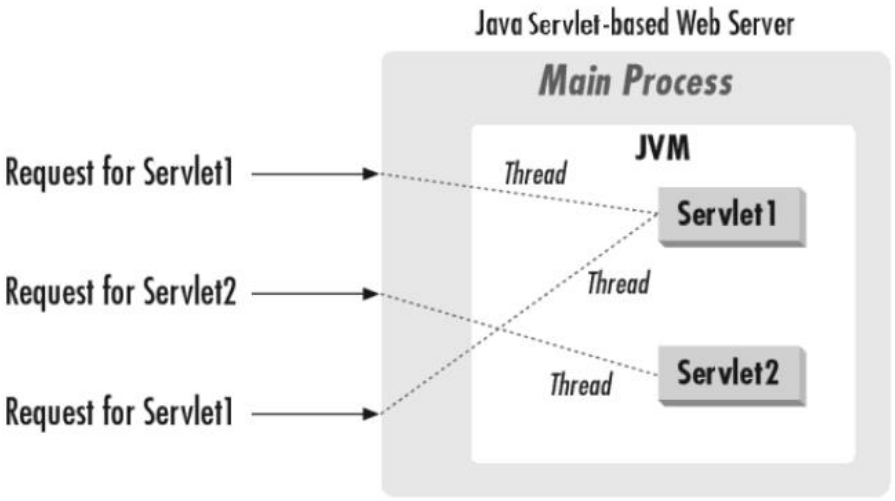

Figura 8: Il life cycle di un servlet

Un servlet viene caricato dal Web server in un particolare ambiente di lavoro chiamato *container*, che si occupa di gestire il suo intero ciclo di vita. Nonostante questo, un servlet ha la possibilità di accedere a tutte le risorse del sistema su cui risiede.

Tutti i servlet ereditano da un supertipo presente nel package "*javax.servlet"* e nel caso degli *HttpServlet*, questi estenderanno in particolare la classe "*javax.servlet.http",* e saranno quindi in grado di comunicare con il client e rispondere dinamicamente alle sue richieste.

4.4.4 JavaServer Pages

Le **JSP**, acronimo di JavaServer Pages, sono tecnologie molto simili dal punto di vista concettuale alle Servlet, ma differiscono da queste per la loro struttura. Infatti una JSP offre la possibilità di inserire all'interno del suo codice, oltre al Java, anche codice HTML puro o mescolato con JavaScript. Per questo motivo spesso vengono utilizzate per lo sviluppo della parte di *presentation layer* all'interno di applicazioni Web,fornendo contenuti dinamici in formato HTML o XML.

Oltre a quanto appena detto, le JSP risultano fortemente correlate alla tecnologia dei servlet poiché nel momento della prima invocazione, le pagine JSP vengono tradotte da un co*mpilatore JSP* in servlet, come illustrato in *Figura 9*. A causa di questa dipendenza

concettuale, anche l'uso delle JSP richiede che sul server Web sia presente un servlet container, oltre al motore JSP che permette anche di compilare le JavaServer Pages. Tomcat ha tutte le tecnologie necessarie per utilizzare sia i servlet che le JSP.

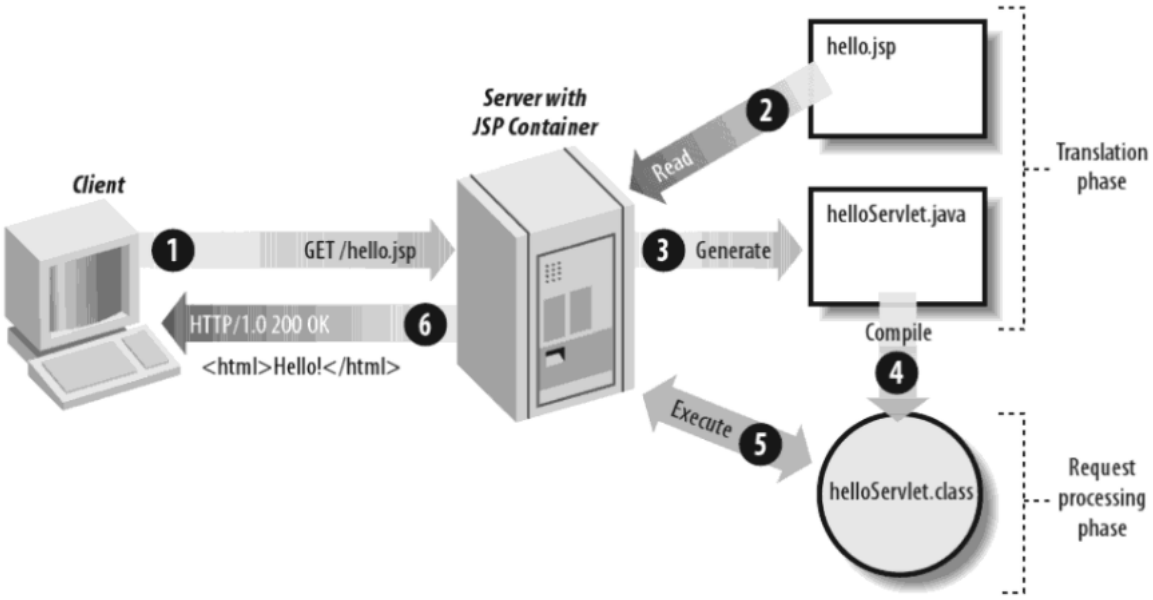

Figura 9: Traduzione da JSP a Servlet

La conversione da JSP a servlet e la successiva compilazione, prende il nome di *translation phase*. Il container inizia la conversione di una pagina JSP automaticamente quando la prima richiesta per quella pagina viene ricevuta dal server. La traduzione può richiedere del tempo e l'utente potrebbe notare un leggero ritardo durante la prima richiesta, per questo motivo la translation phase può avvenire prima che venga effettuata richiesta. Questo modo di procedere viene chiamato *precompilation*. Inoltre fino a quando la pagina JSP rimane invariata, la parte di translation non viene attuata; questo avviene solo nel momento in cui viene fatta una richiesta che modifica il contenuto della pagina. [12]

4.4.5 Java Database Connectivity

Il **JDBC**, termine che indica il Java Database Connectivity, è un'interfaccia scritta in Java, che permette di eseguire istruzioni SQL su un DBMS. Svolge quindi la funzione di *driver*, ovvero di un connettore per database che consente l'accesso e la modifica dei dati indipendentemente dal DBMS utilizzato, come mostra la *Figura 10*.

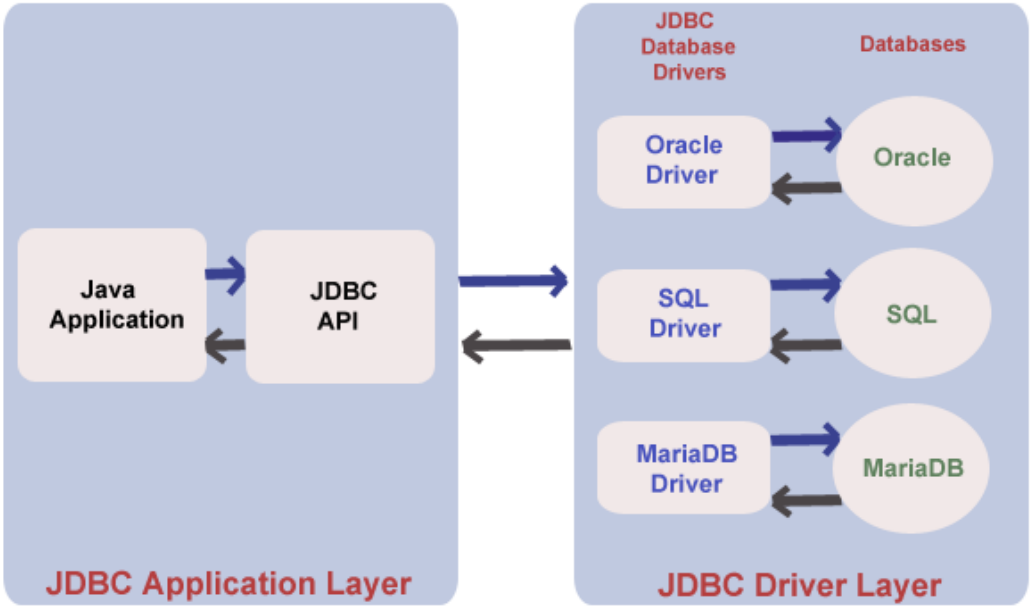

Figura 10: Architettura di JDBC

Il JDBC è costituito da un *API* [16] orientato agli oggetti, che carica dinamicamente i driver appropriati nel JDBC driver manager. Dopo aver effettuato la connessione al database, si possono eseguire delle istruzioni. I comandi SQL come INSERT, UPDATE, DELETE, restituiscono un valore che indica quante tuple sono state coinvolte nell'istruzione. Le query di tipo SELECT restituiscono invece un *ResultSet*, ovvero un oggetto formato da un insieme di tuple e dei metadati che è iterabile tramite il metodo *next()*.

[16] Un *Application Programming Interface* indica un insieme di procedure, ovvero set di definizioni e protocolli con i quali vengono realizzati ed integrati software applicativi

Capitolo 5

Applicazione sviluppata

In questo capitolo verranno descritte in maniera puntuale le pagine che compongono l'applicazione, la base di dati esterna e la JavaServer Page, che insieme permettono di creare un meccanismo di sincronizzazione completo, al fine di soddisfare tutti gli obiettivi definiti nel *capitolo 3.*

Verranno mostrate le pagine dell'App con le relative funzionalità e insieme ad esse verranno mostrati i codici che le generano, per una migliore comprensione della logica di funzionamento.

5.1 Struttura del progetto

Prima di procedere nel descrivere le diverse componenti che formano l'applicazione, è necessario e utile mostrare la sua struttura interna.

Tramite i comandi elencati nel *capitolo 3.1.2*, si avvia lo ***scaffolding*** ovvero quella procedura che automatizza la creazione di alcune interfacce e oggetti. Il termine inglese scaffolding viene infatti tradotto letteralmente come impalcatura e permette di creare quindi quella struttura di base dalla quale partire. Il modo in cui viene organizzata l'App appena creata, rispecchia quanto illustrato nella *Figura 11*, con tre cartelle interne alla cartella principale dell'applicazione "*myApp*" e dei file *.json* .

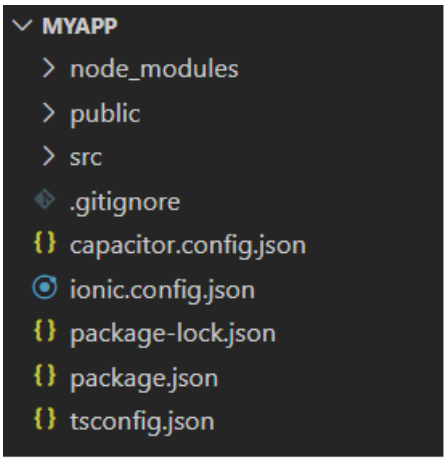

Figura 11: Struttura dell'App al momento della creazione tramite CLI

Una volta che questa struttura è stata creata, è possibile iniziare a costruire le diverse componenti. È possibile generare nuove feature in due modi diversi: in modo manuale, ovvero navigando all'interno della struttura delle directory e creando i diversi oggetti da zero, oppure utilizzando la linea di comando e quindi sfruttando le potenzialità della Ionic CLI. Il framework infatti mette a disposizione il comando "*ionic generate*" per mezzo del quale è possibile generare una varietà di componenti, come moduli, classi, direttive, guard, pagine etc.

La maggior parte del codice sviluppato verrà inserito all'interno della cartella "*src*", la quale contiene i file fondamentali che andranno a realizzare l'App. La struttura di questa cartella è visibile in *Figura 12* e, come mostrato, al suo interno è presente la cartella "*pages"* nella quale saranno presenti tutte le pagine, con annessi fogli di stile, che realizzeranno l'intera applicazione. In questa directory troviamo solo file TypeScript e, per le pagine che lo necessitano, un file CSS dove vengono definite le regole di stile specifiche per una pagina. Il file "*common.css*" è invece un file condiviso da tutte le pagine dell'applicazione e che crea quindi un'interfaccia grafica standardizzata e unificata.

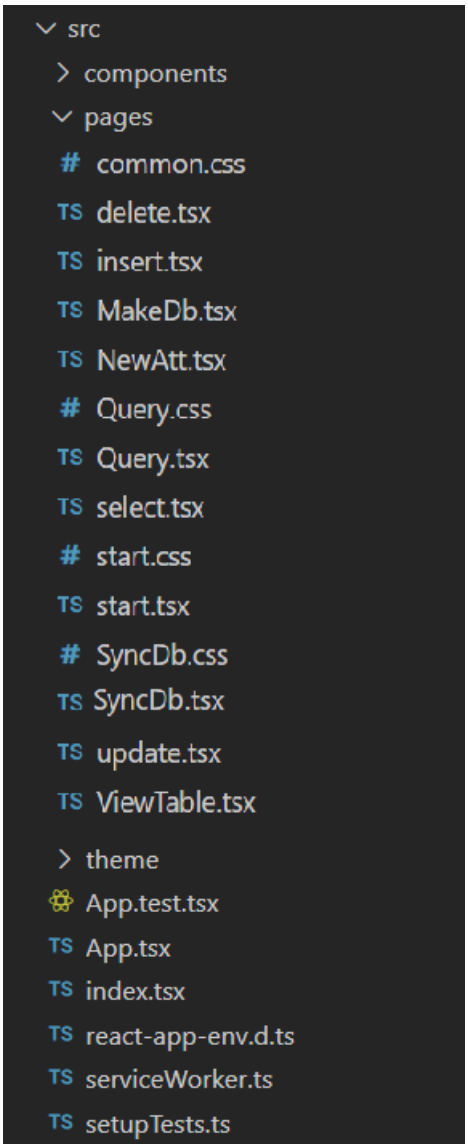

Figura 12: Struttura interna della cartella */src*

Altra cartella fondamentale è la "*public*" mostrata in *Figura 13*:

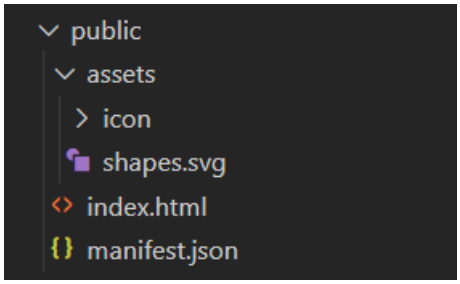

Figura 13: Struttura interna della cartella */public*

Al suo interno è presente il file "*index.html*" che è il punto di ingresso dell'App, ed è utilizzato per attivare l'esecuzione di tutta l'applicazione. Come illustrato nella *Figura 14*, al suo interno si trova il tag *body*, contenente un *div* con ***id = root***.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Ionic App</title>

    <base href="/" />

    <meta name="color-scheme" content="light dark" />
    <meta
      name="viewport"
      content="viewport-fit=cover, width=device-width, initial-scale=1.0, minimum-scale=1.0,
       maximum-scale=1.0, user-scalable=no"
    />
    <meta name="format-detection" content="telephone=no" />
    <meta name="msapplication-tap-highlight" content="no" />

    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />

    <link rel="shortcut icon" type="image/png" href="%PUBLIC_URL%/assets/icon/favicon.png" />

    <!-- add to homescreen for ios -->
    <meta name="apple-mobile-web-app-capable" content="yes" />
    <meta name="apple-mobile-web-app-title" content="Ionic App" />
    <meta name="apple-mobile-web-app-status-bar-style" content="black" />
  </head>

  <body>
    <div id="root"></div>
  </body>

</html>
```

Figura 14: Codice del file *index.html*

Questo tag sarà utilizzato da Ionic React come punto di partenza per la costruzione dell'App. Sarà utilizzato come **nodo radice** del DOM di React: questo framework infatti crea degli elementi React, più semplici e veloci da creare rispetto agli oggetti del DOM del browser, e si prende cura di aggiornare il DOM in modo da essere consistente con gli elementi React.

Le applicazioni solitamente hanno un unico nodo radice e lì vengono renderizzati tutti gli elementi tramite il metodo **ReactDOM.render()** : esso infatti renderizza un elemento

React nel DOM all'interno del container fornito in input e ritorna un riferimento al componente. Se l'elemento React era stato precedentemente renderizzato nel container, verrà eseguito un aggiornamento dell'elemento e verrà aggiornato solo il DOM necessario. [13]

Nel caso di questo progetto, il *ReactDOM.render()* è presente all'interno del file "*/src/index.tsx*". Come visibile in *Figura 15*, la funzione di questo file è quella di renderizzare l'applicazione all'interno del body dell'*index.html* appena descritto.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Figura 15: Codice del file *index.tsx*

L'elemento *<App />*, passato come parametro del metodo, viene creato all'interno del file *"App.tsx*", che verrà descritto nel prossimo capitolo.

5.2 App.tsx

All'interno file *App.tsx* vengono importati diversi tipi di oggetto come mostra la *Figura 16*, numerosi CSS necessari per i componenti della UI di Ionic, icone e le pagine che vanno a comporre l'applicazione.

```
import React from 'react';
import { Redirect, Route } from 'react-router-dom';
import {IonApp, IonIcon, IonLabel, IonRouterOutlet,
        IonTabBar, IonTabButton, IonTabs} from '@ionic/react';
import { IonReactRouter } from '@ionic/react-router';
import { gitCompare, help, server} from 'ionicons/icons';
import MakeDb from './pages/MakeDb';
import Insert from './pages/insert';
import Select from './pages/select';
import Delete from './pages/delete';
import SyncDb from './pages/SyncDb';
import Start from './pages/start';
import Query from './pages/Query';
import Update from './pages/update';
```

```
/* Core CSS required for Ionic components to work properly */
import '@ionic/react/css/core.css';
/* Basic CSS for apps built with Ionic */
import '@ionic/react/css/normalize.css';
import '@ionic/react/css/structure.css';
import '@ionic/react/css/typography.css';
/* Optional CSS utils that can be commented out */
import '@ionic/react/css/padding.css';
import '@ionic/react/css/float-elements.css';
import '@ionic/react/css/text-alignment.css';
import '@ionic/react/css/text-transformation.css';
import '@ionic/react/css/flex-utils.css';
import '@ionic/react/css/display.css';
/* Theme variables */
import './theme/variables.css';
```

Figura 16: Elementi importati nel file *App.tsx*

Successivamente a questi import, c'è l'elemento React "*App*" definito utilizzando la seguente sintassi di React:

```
const App: React.FC = () => (
    .
    .
    .
);
export default App;
```

Una sintassi simile sarà utilizzata per tutte le altre pagine dell'applicazione, sostituendo ad "App" il nome associato alla pagina.

All'interno delle parentesi tonde verranno dichiarate variabili, definite funzioni e definiti i tag HTML e tag Ionic.

Questo file contiene al suo interno due tag unici che lo differenziano da tutte le altre pagine: il tag "<*IonApp*>" e "<*IonReactRouter*>". Quest'ultimo contiene le regole di

routing interne all'App che definiscono in che modo instradare l'utente in base agli input ricevuti.

Come richiesto nella documentazione di Ionic, in ogni App dovrebbe essere presente un singolo componente che svolge questo compito. L'*IonReactRouter* è solo un coordinatore di URL per gli *outlets* di navigazione, che in Ionic sono *IonNav* e *IonTabs*. Questo implica che il router non modifica mai il DOM, quindi non mostra nessun componente ma indica solo a determinati elementi, *IonTabs* nel caso dell'applicazione sviluppata, cosa mostrare[14] .

La *Figura 17* mostra i percorsi definiti per l'App:

```
<IonRouterOutlet>
  <Route path="/makedb" component={MakeDb} exact={true} />
  <Route path="/query" component={Query} exact={true} />
  <Route path="/syncdb" component={SyncDb} exact={true} />

  <Route path="/update" component={Update} exact={true}/>
  <Route path="/insert" component={Insert} exact={true}/>
  <Route path="/delete" component={Delete} exact={true}/>
  <Route path="/select" component={Select} exact={true} />

  <Route path="/start" component={Start} exact={true} />

  <Route path="/" render={() => <Redirect to="/start" />} exact={true} />
</IonRouterOutlet>
```

Figura 17: Percorsi di routing interni all'App

I primi tre, ovvero "*makedb*" "*query*" e "*syncdb*", sono i percorsi che vengono definiti per i tre tab presenti nella parte bassa dell'applicazione. Questi permettono di muoversi tra le tre differenti pagine dell'applicazione e sono sempre visibili (quindi cliccabili) all'utente. I quattro path successivi sono invece relativi ai quattro bottoni che si trovano nella pagina "*query*". Gli ultimi due tag sono invece relativi al carousel che appare quando viene aperta l'applicazione: il primo specifica la pagina dove sono definiti gli elementi che lo compongono, il secondo tag serve per indicare la pagina dove viene rediretto l'utente nel momento in cui apre l'App e quindi non è definito un path specifico.

Nella seconda parte del file *App.tsx* è invece definita la struttura dei **tab**, ovvero tre tab cliccabili che permettono di navigare nelle tre diverse pagine principali dell'applicazione. La *Figura 18* illustra il codice di questi elementi:

```
<IonTabBar slot="bottom">
  <IonTabButton tab="makedb" href="/makedb">
    <IonIcon icon={gitCompare} />
    <IonLabel>Make /delete DB</IonLabel>
  </IonTabButton>

  <IonTabButton tab="query" href="/query">
    <IonIcon icon={help} />
    <IonLabel>Make a Query</IonLabel>
  </IonTabButton>

  <IonTabButton tab="syncdb" href="/syncdb">
    <IonIcon icon={server} />
    <IonLabel>Sync Db</IonLabel>
  </IonTabButton>
</IonTabBar>
```

Figura 18: Codice che definisce i tre tab

Come si può vedere la struttura risulta composta da diversi tag nidificati: il tag *IonTabBar* che incorpora l'intera struttura e grazie al valore dell'attributo "*slot*" si posiziona nella parte bassa dell'App, mentre ogni singolo tab viene definito all'interno del tag *IonTabButton*. Quest'ultimo ha come attributo il riferimento di pagina verso la quale sarà rediretto l'utente al momento del click su di esso, oltre ad avere nidificati due ulteriori tag. *IonIcon* è un tab vuoto e serve a definire l'icona che apparirà all'interno di ciascun tab, mentre *IonLabel* imposta la scritta che verrà visualizzata nel singolo tab.

5.3 Carousel

Come scritto nel capitolo precedente, al momento dell'avvio dell'applicazione, sarà mostrato un carousel all'utente. Questo **slide show** è definito all'interno del file *start.tsx* ed il codice che lo crea è definito unicamente da tag Ionic. Il framework si occupa sia della parte grafica che della logica che permette lo scorrere delle slides al tocco dell'utente.

Il codice che permette la creazione di questo componente è visibile in *Figura 19*:

```
<IonSlides>
    <IonSlide>
      <div className="slide">
          <img src="assets/img/img1.png"/>
          <h1>Benvenuto</h1>
          <p> Crea il tuo
            <b>Database Locale</b> cliccando semplicemente sul bottone</p>
      </div>
    </IonSlide>
    <IonSlide>
      <img src="assets/img/img2.png"/>
      <h1>Interegisci col DB</h1>
      <p>Puoi fare differenti <b>query</b>, sarà tutto facile e intuitivo</p>
    </IonSlide>
    <IonSlide>
      <img src="assets/img/img3.png"/>
      <h1>Sincronizza il DB</h1>
      <p>Sincronizza la tua applicazione con il <b>Database esterno</b>:
      basta premere il bottone e aspettare che il caricamento finisca</p>
      </IonSlide>
    <IonSlide>
      <img src="assets/img/img4.png"/>
      <h1>Inizia subito</h1>
      <IonButton fill="clear" href="/MakeDB">Continue
      <IonIcon slot="end" name="arrow-forward"></IonIcon>
    </IonButton>
    </IonSlide>
</IonSlides>
```

Figura 19: Codice che definisce il carousel

In Ionic il componente dello slides show viene creato come contenitore a più sezioni, e contiene un numero di slides non limitato. Utilizza le tecnologie di **Swiper.js**, che viene ritenuto il framework più moderno per realizzare le tecnologie di *swipe* (scorrimento).[15]

Ciascuna slide viene definita e modificata all'interno del tag *IonSlide*. Sono state codificate in modo tale da mostrare un'immagine centrata con al di sotto un testo che introduce ciò che l'utente potrà realizzare. Ogni slide corrisponde ad una pagina

dell'applicazione, per questo motivo e per facilitare l'utilizzo allo user, l'immagine presente sulla slide è stata ricreata a partire dall'interfaccia grafica della pagina dell'App.

Alla fine dello slides show è presente un'ultima slide, corrispondente all'ultimo tag *IonSlide*, che permette di accedere finalmente all'applicazione vera e propria. È infatti definito un *IonButton*, ovvero un bottone, che ha un collegamento ad una pagina dell'App. Da notare come in questo file non sia definita nessuna regola di routing ma, come descritto nel capitolo precedente, è tutto definito all'interno del file *App.tsx* .

La *Figura 20* mostra il risultato grafico del codice appena illustrato:

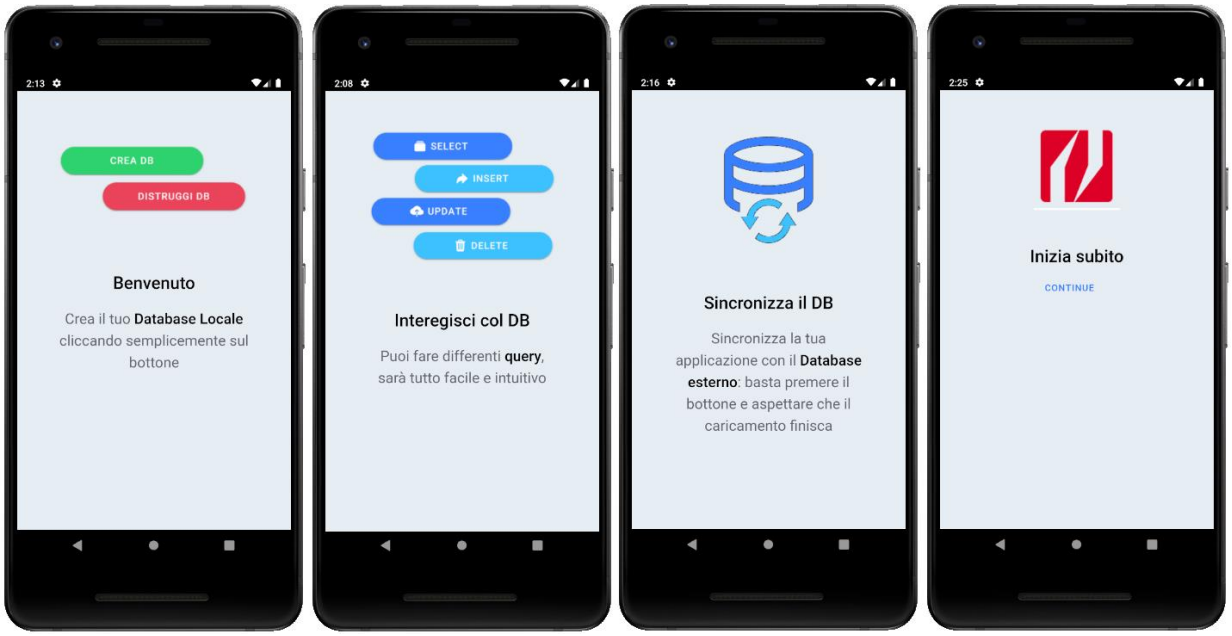

Figura 20: Interfaccia grafica dello slides show

5.4 Interfaccia per la creazione del database

Passiamo ora ad analizzare quella che è la vera e propria applicazione, ovvero tutte quelle componenti che possono essere utilizzate dall'utente. Partiamo quindi dall'analizzare la prima pagina, vale a dire quella pagina che viene renderizzata una volta che lo slide show è stato chiuso, oppure alla quale si può accedere tramite il tab in basso a sinistra. Questa risulta formata da tre bottoni, che attivano tre differenti metodi.

Iniziamo dall'analizzare il codice che produce la parte grafica, illustrata in *Figura 21,* presente all'interno del file denominato "***MakeDb.tsx***":

Figura 21: Interfaccia grafica della pagina *MakeDb.tsx*

Da questa Figura si vede come è strutturata una pagina dell'applicazione: un header contenente il titolo, una parte centrale e in basso i tre tabs. Quest'ultimi sono stati già descritti nel *capitolo 5.2* e non verranno trattati nuovamente; oltretutto il codice non è presente in questo file in quanto presente solo all'interno del file "*App.tsx*".

L'header è realizzato tramite tag Ionic e il suo codice è visibile nella *Figura 22* seguente:

```
<IonHeader>
  <IonToolbar>
    <IonTitle>Crea o distruggi il Database</IonTitle>
  </IonToolbar>
</IonHeader>
<IonContent fullscreen>
  <IonHeader collapse="condense">
    <IonToolbar>
      <IonTitle size="large">Crea o distruggi il Database</IonTitle>
    </IonToolbar>
  </IonHeader>
```

Figura 22: Codice relativo all'header di *MakeDb.tsx*

Questo insieme di tag nidificati è predefinito in Ionic e la scritta che successivamente viene resa visibile all'utente è contenuta all'interno del tag *IonTitle*. Questa struttura dell'header è la stessa per tutte le pagine e per questo non verrà descritta nuovamente; ciò che varia è la scritta presente all'interno del tag appena citato.

Il codice che permette di realizzare il resto del contenuto del file, ovvero i tre bottoni con annessi testi, è descritto in *Figura 23*:

```
<section className="makeDbButton">
    <h1 className="title">CREA IL DATABASE </h1>
    <p>
      Premendo il bottone verrà creato un database con le tabelle già popolate
    </p>
    <IonButton shape="round" size="large" color="success" className ="button center" onClick={makeDb} >Crea DB </IonButton>

</section>
<section className="makeDbButton">
    <h1 className="title">DISTRUGGI IL DATABASE</h1>
    <p>
      Premendo il bottone verranno eliminate tutte le tabelle del DB
    </p>
    <IonButton shape="round" size="large" color="danger" className ="button center" onClick={destroyDb} >Distruggi DB</IonButton>
</section>
<section className="makeDbButton">
    <h1 className="title">CREA DAL DB ESTERNO</h1>
    <p>
      Premendo il bottone varrà creato un database, inizianizzandolo dal DB esterno
    </p>
    <IonButton shape="round" size="large"  className ="button center" onClick={makeFromExtDb} >Crea DB</IonButton>
</section>
```

Figura 23: Codice relativo al contenuto del file *MakeDb.tsx*

Anche in questo caso si può notare una struttura divisa in tre, ciascuna corrispondente ad un bottone. Ogni parte è delimitata dal tag HTML *section,* che a differenza dell'HTML classico non utilizza l'attributo *class* per definire una classe ma *className*; all'interno della sezione è poi presente un titolo <*h1*>, seguito da un testo <*p*> e infine un bottone. Si nota dalla differente colorazione che vi è una separazione tra i tag relativi all'HTML,

che l'editor di testo Visual Studio Code colora di blu, e quelli che sono originari di Ionic React, che vengono invece colorati di verde.

Il bottone è definito dal tag Ionic IonButton, il quale presenta diversi attributi: *shape, size* e *color* hanno dei valori che sono predefiniti da Ionic, ad esempio impostando il colore del bottone uguale a *success* si avrà graficamente un colore verde. Infine troviamo l'attributo *onClick* che permette di attivare una funzione al click dell'utente sul bottone; a differenza della sintassi classica che prevede il passaggio della funzione da attivare come stringa racchiusa tra virgolette, la sintassi del *tsx* obbliga a passare l'event handler come espressione React. Questa viene racchiusa tra parentesi graffe e successivamente alla compilazione le espressioni diventano normali chiamate a funzioni JavaScript.

Terminata l'analisi relativa alla parte grafica, possiamo ora trattare il codice e le funzioni relative ai bottoni appena descritti. Iniziamo della prima funzione, ovvero ***makeDB***, il cui codice è stato riportato nella *Figura 24*:

```
{/* Funzione che crea e popola un nuovo db */}
function makeDb(){
  SQLite.create({ name: 'dbprova.db', location: 'default'})
        .then(db => {
          db.executeSql('CREATE TABLE IF NOT EXISTS master(order_id INT PRIMARY KEY, client VARCHAR(30), client_id INT, order_date DATE, rowstatus CHAR, last_update
            .then(() => {
              console.log('Creata tabella master');
              let insert : string = "INSERT INTO master( order_id, client, client_id, order_date, rowstatus, last_update) VALUES";
              insertDB(db, tabMast, insert);
            })
            .catch(e => console.log(e))

          db.executeSql("CREATE TABLE IF NOT EXISTS details(order_id INT, order_num INT(4) , article VARCHAR(30), article_id INT(5), long_desc VARCHAR(200), quantity
            .then(() => {
              console.log('Creata tabella details');
              let insert = "INSERT INTO details(order_id, order_num, article, article_id, long_desc, quantity, rowstatus, last_update) VALUES";
              insertDB(db, tabDetails, insert);

            })
            .catch(e => console.log(e))

          db.executeSql("CREATE TABLE IF NOT EXISTS last_sync(last_update TIMESTAMP PRIMARY KEY )",[])
          .then(()=>{
            db.executeSql("INSERT INTO last_sync(last_update) VALUES (strftime('%Y-%m-%d %H:%M:%S','now'))",[])
          })
          .then(()=>console.log("Inserito last update"))
          .catch(e => console.log(e))
          })

         .then(e => alert("Database creato e popolato"))
         .catch(e => console.log(e))
};
```

Figura 24: Codice della funzione *makeDb*

Per mezzo di questa funzione è possibile creare un nuovo database, creare tre tabelle interne allo stesso e infine riempire le tabelle con dei dati di prova già presenti all'interno

del codice dell'applicazione. Analizziamo ora più nel dettaglio le componenti principali di questa funzione. Prima di tutti è necessario creare la base di dati interna e questo si può fare utilizzando la funzione predefinita ***SQLite.create()***, alla quale verranno passati come parametri il nome del database da creare (in questo caso *dbprova.db*) e la collocazione all'interno della struttura dell'applicazione (per comodità verrà lasciato il percorso di default). Questa funzione verrà utilizzata in tutte le altre procedure dove sarà necessario accedere al database, infatti essa non permette solo di creare una base di dati ma anche di aprirne uno già creato.

Prima di procedere e spiegare codice successivo, è necessario introdurre un nuovo concetto, ovvero le ***Promise***. Questi oggetti sono collegati alla programmazione asincrona: una funzione che lavora in questa maniera non blocca il restante codice JavaScript fino al suo completamento, ma agisce in background all'interno del programma. È necessario utilizzare questa tipologia di programmazione ogni volta che si ha una procedura che per essere eseguita necessita che un'altra funzione abbia terminato la sua esecuzione. La funzione sviluppata ricade in questa casistica: infatti non è possibile creare una tabella in database, se prima questo non è stato aperto o creato. Una promise rappresenta quindi un'operazione che non è ancora completata, ma potrebbe esserlo in futuro. Queste possono quindi presentarsi in uno dei seguenti stati:

- ***Pending***, è lo stato iniziale in cui non è né soddisfatta né respinta e si trova quindi in attesa
- ***Fulfilled***, significa che l'operazione si è conclusa con successo
- ***Rejected***, significa che l'operazione è fallita

Quindi una promise in *pending* può evolvere in *fulfilled* con un valore, oppure in *rejected* con una motivazione ovvero un errore. [16]

Quando accade una di queste due situazioni, vengono chiamati gli handler associati che sono accodati dal metodo ***then()*** nel caso in cui la promise sia stata soddisfatta, oppure dal metodo ***catch()*** nel caso in cui si sia verificato un errore.

Tornando quindi alla funzione della *Figura 24*, nel caso in cui la creazione del database sia andata a buon fine, si entrerà all'interno della funzione *then*, la quale utilizzerà l'oggetto restituito dalla funzione precedente, chiamato *db*, che rappresenta il database appena creato. Tramite la funzione ***executeSql()***, è possibile eseguire una query al database. In questo caso verrà creata una tabella *master* con gli attributi elencati nel codice. Anche in questo caso ci troviamo di fronte alla tecnologia delle promise ed è quindi necessario utilizzare la funzione *then* per procedere. Nel caso in cui la promise dia un errore, si entrerà all'interno della funzione *catch* che tramite il *console.log* permetterà di visualizzare l'errore verificatosi. Se dunque anche la creazione della tabella va a buon fine, si può proseguire con il riempimento della stessa. Questo avviene tramite la chiamata di una funzione esterna denominata ***insertDB***, alla quale vengono passati tre parametri: l'oggetto che rappresenta il database, un array contenente tutti i dati di prova da immette all'interno della tabella e una stringa che compone la parte iniziale di ogni query che dovrà essere eseguita per l'immissione dei dati. Il codice è rappresentato in *Figura 25*:

```
{/** Funzione necessaria per riempire il DB insert per insert */}
 function insertDB(db: any, array: string[], insert: string){
  const promises =[];
  for (var i=0; i<array.length; i++){
    promises.push(db.executeSql(insert + array[i]+",'i', strftime('%Y-%m-%d %H:%M:%S','now'))",[]));
  }
  Promise.all(promises)
    .then( e => console.log(e))
    .catch(e => console.log(e))
};
```

Figura 25: Codice della funzione *insertDB*

Tramite questa funzione viene riempita una tabella con tutti i dati. Per ciascun elemento dell'array contenente i dati da inserire, verrà effettuata una query. Ma poiché la funzione *executeSql* utilizza le promise, è necessario utilizzare un costrutto aggiuntivo: all'inizio della funzione viene creato un array chiamato ***promises*** che verrà riempito, per mezzo del metodo ***push()*** specifico per gli array, con tante promise quanti sono gli elementi dell'array passato come parametro alla funzione.

Infine viene eseguito il metodo ***Promise.all()*** al quale viene passato come parametro l'array di promise, e questo attiverà la procedura *then* solo e se tutte le promise vengono soddisfatte.

All'interno della funzione *makeDb* questa procedura viene ripetuta per tutte e tre le tabelle che comporranno la base di dati: si crea la tabella tramite una query e se questa va a buon fine, si richiama la funzione appena descritta *insertDB* per riempire la tabella con i dati.

Ora verrà descritta la funzione relativo al secondo bottone, ovvero quello che permette di azzerare tutti i dati contenuti nel database. È bene specificare che questa funzione in un contesto reale non dovrebbe poter essere attivabile da parte dell'utente, ma in questo caso è stata inserita per un testing completo delle funzionalità dell'App. La seguente *Figura 26* descrive il codice che permette di effettuare questa operazione:

```
function destroyDb(){
  SQLite.deleteDatabase({ name: 'dbprova.db', location: 'default'})
  .then(() => {
    alert("Database Eliminato");
    console.log("Database eliminato");
  });
};
```

Figura 26: Codice relativo alla funzione *destroyDb*

La figura mostra come questa procedura sia facilmente realizzabile grazie al metodo predefinito di SQLite ***deleteDatabase***, il quale richiede di specificare il nome della base di dati da eliminare e il percorso interno alla struttura dell'applicazione. Se questa operazione va a buon fine, grazie alla funzione *alert,* verrà mostrato un pop-up all'utente che lo aggiorna della corretta eliminazione del database.

Infine la funzione relativa al terzo e ultimo bottone ha un obiettivo simile alla prima funzione ma lo raggiunge mediante strumenti diversi. La struttura della procedura che si attiverà al click del bottone è quella mostrata in *Figura 27:*

```
function makeFromExtDb(){
  SQLite.create({ name: 'dbprova.db', location: 'default'})
        .then(db => {
          db.executeSql('CREATE TABLE IF NOT EXISTS master(order_id INT PRIMARY KEY, client VARCHAR(30), client_id
            .then(() => {
              console.log('Creata tabella master');
              requestToExtDb('master');
            })
            .catch(e => console.log(e))

          db.executeSql("CREATE TABLE IF NOT EXISTS details(order_id INT, order_num INT(4) , article VARCHAR(30),
            .then(() => {
              console.log('Creata tabella details');
              requestToExtDb('details');
            })
            .catch(e => console.log(e))

          db.executeSql("CREATE TABLE IF NOT EXISTS last_sync(last_update TIMESTAMP PRIMARY KEY )",[])
          .then(()=>{
            db.executeSql("INSERT INTO last_sync(last_update) VALUES (strftime('%Y-%m-%d %H:%M:%S','now'))",[])
          })
          .then(()=>console.log("Inserito last update"))
          .catch(e => console.log(e))
          })

         .then(e => alert("Database creato e popolato"))
         .catch(e => console.log(e))
}
```

Figura 27: Codice relativo alla funzione *makeFromExtDb*

Se si confronta quest'ultima figura con la *Figura 24* relativa al primo bottone, si noteranno molte analogie. L'unica parte di codice che cambia è relativa alla funzione che serve per riempire le tabelle create con dei dati. Mentre nel caso precedente questo veniva realizzato utilizzando dati preimpostati all'interno dell'App, in questo caso si richiama la funzione *requestToExtDb* che si collegherà con il database esterno.

Per ricevere i dati provenienti dalla base di dati esterna, l'applicazione dovrà prima fare una richiesta GET al Web server. Nella programmazione in JavaScript questo è ottenibile in diversi modi: tramite oggetti *XMLHttpRequest* che permettono di interagire con server, recuperando i dati da un URL senza dover aggiornare l'intera pagina e per questo motivo è ampiamente utilizzato nella programmazione AJAX. Un modo più flessibile di ottenere questo risultato è attraverso la *Fetch API,* che mette a disposizione un'interfaccia per manipolare parti della HTTP pipeline, sia per richieste che per risposte; oltretutto fornisce il metodo *fetch()* che permette di ottenere facilmente risorse in modo asincrono. [17]

Una ulteriore possibilità, utilizzata in questo progetto, è data dall'uso della libreria JavaScript ***Axios.js***. Quest'ultima mette a disposizione dei metodi che permettono di effettuare differenti tipi di richieste in modo semplice e flessibile. La *Figura 28* illustra

la funzione ***requestToExtDb*** che viene utilizzata per fare una richiesta al server e ricevere i dati da immettere nella base di dati dell'applicazione. È possibile fare una richiesta di tipo GET utilizzando la funzione ***get()*** che mette a disposizione la libreria Axios, dando come parametro l'URL del Web server: viene indicato l'IP e la porta alla quale è possibile collegarsi (8080 è la porta di ascolto di Tomcat), e poi si specifica il percorso fino al file desiderato, che verrà esaminato successivamente, e si passa come valore del parametro *table* il nome della tabella dalla quale si vogliono recuperare i dati.

```
function requestToExtDb(table:string){
  axios.get("http://192.168.11.246:8080/prova/dbGetData.jsp?table="+table)
      .then(response => response.data)
      .then(data =>{
        let query : string ="INSERT INTO "+table+"(";
        let values : string ="";
        let flag : boolean = true;

        data.map((row:any)=>{
          values += " (";
          for (var key in row){
            {/* Il flag serve per selezionare i parametri solo per la prima iterazione */}
            if(flag) query += " "+key+",";
            if(key=="rowstatus") values += "'i',";
            else if(key=="last_update") values += " strftime('%Y-%m-%d %H:%M:%S','now'),";
            else if(!isNaN(row[key])){
              values+= " "+row[key]+",";
           } else values+= " '"+row[key]+"',";
          }
          values=values.slice(0,-1);
          values += "),";
          flag=false;
        })
        query=query.slice(0,-1);
        values=values.slice(0,-1);
        SQLite.create({ name: 'dbprova.db', location: 'default'})
         .then(db => {
            db.executeSql(query+") VALUES "+ values,[])
         })
         .catch(e => console.log(e))
      })
}
```

Figura 28: Codice relativo alla funzione *requestToExtDb*

Anche nel caso di Axios si effettuano richieste in modo asincrono tramite le *promise* e per questo si utilizzano i metodi *then* e *catch* precedentemente introdotti. La risposta del Web server è un array JSON. Ciascun elemento dell'array corrisponderà ad una tupla della tabella, con un certo numero di coppie "nome:valore", pari al numero di colonne

della tabella stessa. Questa risposta così composta, dovrà essere destrutturata dall'applicazione per estrarne i dati.

Successivamente alla richiesta GET, quando la promise verrà soddisfatta, si entrerà nel primo *then* che estrarrà i dati dal corpo della risposta tramite il *"response.data*"; questi dati saranno strutturati come descritto e sarà necessario quindi iterare all'interno dell'array di JSON. Ciò è ottenuto attraverso l'utilizzo del metodo ***map()***, che si può ritenere come un ciclo for ma specifico per gli array. La variabile *row* identifica ad ogni nuova iterazione l'elemento dell'array considerato, che nel nostro caso è un JSON che contiene i dati relativi ad una tupla della tabella. È necessario impostare una seconda iterazione intera alla prima, in quanto bisogna estrarre i valori di tutti gli elementi del JSON. L'obiettivo di questa parte di funzione è creare una stringa corrispondente alla query per l'inserimento dei dati, che verrà successivamente eseguita, strutturata nel seguente modo:

INSERT INTO ***tabella (nomi colonne)*** **VALUES (** ***valori) , (valori) , (valori)***

Ovvero si creerà un'unica query, che per la sintassi di SQLite necessita di specificare tutti i nomi delle colonne anche nel momento dell'inserimento, e saranno scritte tante coppie di parentesi per quanti sono i valori da inserire.

Quanto appena scritto, spiega le variabili inserite all'interno del codice della funzione: la variabile *query* verrà composta in modo tale da contenere tutta la prima stringa che va a formare la query "INSERT INTO *tabella (nomi colonne)*", infatti la variabile *flag* è necessaria per aggiungere le stringhe corrispondenti ai nomi delle colonne alla stringa della variabile *query* solo nella prima iterazione del *map*. Il restante codice interno al ciclo for crea la stringa corrispondente alla variabile *values*, che specifica i valori da inserire all'interno della query. Sono presenti due if che modificano due valori dei campi delle tuple ricevute dalla base di dati esterna, corrispondenti alle colonne "*rowstatus*" e "*last_update*", cambiandoli con i valori "*i*" per la prima e la data e ora del momento dell'inserimento per la seconda.

Il successivo costrutto else-if è necessario per strutturate la stringa in base al fatto che il valore da inserire sia una stringa/data oppure un numero. Nel primo caso infatti è necessario racchiudere il valore all'interno delle virgolette, nel secondo caso no. Una differente sintassi, quindi immettendo le virgolette in entrambi i casi o in nessuno dei due, porterebbe ad un errore di inserimento.

Si può notare infine che viene utilizzato il metodo ***slice()*** che risulta necessario in quanto strutturando le stringhe all'interno di un ciclo, l'ultima iterazione lascia come carattere delle variabili *query* e *values* una virgola che se non eliminata dalla stringa porterà ad un errore durante la query al database.

Alla fine di questa procedura viene creata una stringa ben strutturata, che permette di inserire tuple correttamente valorizzate nella tabella del database interno, utilizzando i metodi *SQLite.create()* e successivamente *executeSql()* già descritti nelle precedenti funzioni.

5.5 Interfaccia per la modifica del database interno

In questo capitolo si tratterà la pagina dell'applicazione che corrisponde al tab centrale, dalla quale è possibile scegliere quale tipologia di query effettuare sul database interno. L'utente si troverà di fronte a quattro bottoni, come illustrato in *Figura 29*, ognuno collegato ad una differente pagina.

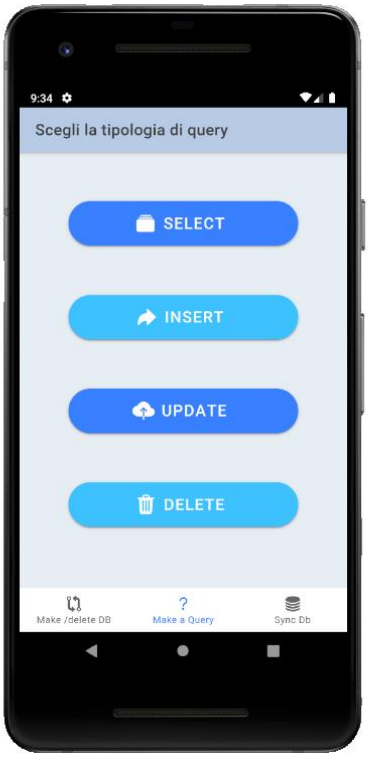

Figura 29: Interfaccia grafica della pagina *query.tsx*

L'interfaccia grafica presenta quattro bottoni colorati in modo alterno, ciascuno dei quali presenta un'icona interna che aiuta l'utente della scelta della query da effettuare. Si è scelto di riportare in *Figura 30* solo il codice relativo a questi bottoni tralasciando gli *import* e altri tag che contengono questo codice in quanto, come chiarito per la *Figura 22*, questo risulterebbe ripetitivo. Come mostrato in figura, la struttura che forma quest'interfaccia è divisa in quattro parti uguali nei tag, che si differenziano solo per i valori di alcuni attributi. Da notare il tag *IonIcon* che permette di inserire un'icona, scelta da un set di icone open source sviluppate dal team di Ionic, che tramite l'attributo *slot* valorizzato con "start" viene posizionato nella parte sinistra del bottone.

```
<IonButton shape="round" size="large" className ="queryButton center" href="/select" >
  <IonIcon slot="start" icon={albums} />
  <IonLabel>Select</IonLabel>
</IonButton>
<IonButton shape="round" size="large" color="secondary" className ="queryButton center" href="/insert" >
  <IonIcon slot="start" icon={arrowRedo} />
  <IonLabel>Insert</IonLabel>
</IonButton>
<IonButton shape="round" size="large" className ="queryButton center" href="/update" >
  <IonIcon slot="start" icon={cloudUpload} />
  <IonLabel>Update</IonLabel>
</IonButton>
<IonButton shape="round" size="large" color="secondary" className ="queryButton center" href="/delete" >
  <IonIcon slot="start" icon={trash} />
  <IonLabel>Delete</IonLabel>
</IonButton>
```

Figura 30: Codice che crea i bottoni di *query.tsx*

Infine è presente il tag *IonLabel* che etichetta il bottone con un testo interno. Ciascun bottone ha un attributo *href* che lo collega ad un'altra pagina dell'applicazione; in questo caso sono collegati alle pagine che permettono di effettuare la query descritta nell'*IonLabel*.

Di queste pagine verrà descritta di seguito solo quella relativa alla query SELECT, in quanto il codice risulterebbe in molti aspetti ripetitivo, ma verranno comunque descritte le differenze di codice presenti.

5.6 Interfaccia per l'esecuzione di una query al database interno

Nel caso in cui l'utente clicchi sul primo bottone della pagina appena descritta, ovvero quello etichettato con "select", verrà rediretto su un'altra pagina che gli permetterà di eseguire una query per selezionare delle tuple del database interno.

L'interfaccia che lo user si troverà di fronte è mostrata nella *Figura 31*, ed è formata dal primo campo, non modificabile, che definisce l'operatore e a seguire tre componenti modificabili: un elemento che permette di scegliere la tabella tra quelle inserite all'interno del database, un corpo nel quale si possono inserire oppure no i valori dei campi per filtrare la select e infine un bottone tramite il quale è possibile effettuare la query.

Figura 31: Interfaccia grafica della pagina *select.tsx*

Il componente che permette di scegliere la tabella è similare alla tipologia di input denominata “radio” nell’HTML classico: definisce singoli elementi di una lista analoghi a delle checkbox, ma selezionabili in mutua esclusione. In questo progetto Ionic dispone di tag personalizzati, visibili in *Figura 32*, ovvero *IonSelect* usato come contenitore degli elementi e *IonSelectOption* che identifica ciascun elemento.

```
<IonItem>
  <IonLabel>Seleziona tabella dove prelevare i dati</IonLabel>
  <IonSelect value={table} placeholder="Choose the table" onIonChange={e => {setTable(e.detail.value); chooseValues()}}>
    <IonSelectOption value="master">Master</IonSelectOption>
    <IonSelectOption value="details">Details</IonSelectOption>
  </IonSelect>
</IonItem>
```

Figura 32: Codice che permette di selezionare quale tabella scegliere

Prima di procedere ad analizzare gli attributi interni al tag *IonSelect*, risulta opportuno spiegare un particolare elemento di React, ovvero gli *hooks*.

Un ***hook*** è una tecnologia introdotta recentemente da React, che viene adoperata per utilizzare lo *state*[17] ed altre funzionalità proprie di React senza dover scrivere una classe. La funzione *useState* è un *hook* e viene richiamato all'interno di un componente per aggiungere uno stato locale. Lo *useState* restituisce una coppia: il valore dello *state* corrente ed una funzione che permette di aggiornarlo. In questo caso il codice è quello mostrato nella *Figura 33*:

```
const [table, setTable] = useState<string>('master');
```

Figura 33: Codice esplicativo dell'hook *useState*

Da questa figura si deduce come lo *useState* definisca il tipo di dato, tra parentesi angolate, e un valore assunto di default, scritto tra parentesi tonde. Le due variabili tra le parentesi quadre invece sono rispettivamente la variabile che assume un determinato stato e la funzione che deve essere utilizzata per modificare quello stato.

Fatta questa precisazione, si può tornare ad analizzare il codice relativo alla precedente *Figura 32*. Il tag *IonSelect* ha definiti al suo interno i seguenti attributi: *value* che assume il valore scelto tra le opzioni selezionabili, un *placeholder* che sarà visibile come titolo della finestra che si aprirà quando l'utente cliccherà su quel campo per scegliere una delle opzioni, ed infine l'attributo *onIonChange* al quale viene assegnata un'espressione. Nel caso in cui l'utente vari la selezione di quel campo, quindi scelga una tabella diversa per la quale effettuare la query, viene captato l'evento "*e*" e viene settato un diverso valore della variabile *table* tramite la funzione *setTable*; successivamente viene triggerato il metodo *chooseValues()* il quale utilizza le funzionalità di React per renderizzare dei tag Ionic che andranno a creare la parte di codice successiva che mostra gli attributi della tabella con vicino i campi di input. Per non appesantire eccessivamente la trattazione ed essere ripetitivi nelle spiegazioni non si mostra il codice relativo a questa funzione in quanto semplicemente, in base alla tabella selezionata, restituisce per mezzo del *return*

[17] Lo *state* è un oggetto integrato nelle componenti di React nel quale vengono memorizzati i valori delle proprietà che appartengono ad un componente; quando lo stato dell'oggetto cambia, il componente viene nuovamente renderizzato

un insieme di tag Ionic annidati come *IonItem*, *IonLabel*, *IonInput* che andranno a formare l'interfaccia grafica della *Figura 31* vista precedentemente.

Non si mostra neanche il codice relativo al bottone in quanto già ampiamente trattato nei capitoli precedente, ma è interessante analizzare la funzione che si attiva al momento del click, ovvero la ***executeQuery()***. Questa va a prendere gli input inseriti dall'utente e crea una stringa correttamente formattata, in modo simile a quanto illustrato nella *Figura 28*. Nella seconda parte della funzione, mostrata nella *Figura 34*, viene eseguita la query al database e vengono create dinamicamente delle strutture che permettono di visualizzare il risultato della query.

```
  query += "SELECT * FROM "+table+" WHERE "+ where;
}
SQLite.create({ name: 'dbprova.db', location: 'default'})
      .then(db => {
        db.executeSql(query,[])
        .then(data => {
            console.log("Query eseguita")
            let keys : any[] = [];
            {/* Estrae tutti i nomi degli attributi */}
            Object.keys(data.rows.item(0)).map(key=>{
              keys.push(key);
            })
            for( var i=0; i < data.rows.length; i++){
              queryData.push(data.rows.item(i));
              console.log(data.rows.item(i));
            }
            cardCreator(queryData,keys);
          }

        )
        .catch(()=>setShowWrongQueryAlert(true))
      })
```

Figura 34: Parte di codice della funzione *executeQuery*

La prima linea di codice della figura mostra la query che, dopo opportuni passaggi, viene creata e formattata; successivamente viene aperta la connessione con il database interno tramite la funzione *SQLite.create()* ed eseguita la query tramite l'*executeSql*. Successivamente viene inizializzato l'array *keys* che verrà riempito con tutti i nomi delle colonne e questo servirà per la successiva visualizzazione a schermo del risultato della query. L'estrazione avviene in questa maniera: viene preso il primo elemento del risultato

della query, ovvero la prima tupla, che corrisponde a *data.rows.item(0)* e tramite il metodo predefinito *Object.keys()* vengono estratte tutte le chiavi ovvero i nomi delle colonne; successivamente si itera l'insieme delle keys e ognuna viene messa all'interno dell'array *keys* precedentemente inizializzato tramite il metodo *push()*.

Dopo aver riempito con i nomi delle colonne l'array *keys*, il codice prosegue con un ciclo for nel quale viene riempito l'array *queryData* con le tuple della tabella, corrispondenti all'oggetto *data.rows.item(i)*, creando così un array multidimensionale.

Infine viene fatta una chiamata alla funzione *cardCreator* alla quale vengono passati come parametri l'array multidimensionale appena creato *queryData* e l'array contenente tutti i nomi delle colonne *keys*. Il risultato di questa procedura è mostrato nella *Figura 35*:

Figura 35: Risultato grafico dell'esecuzione della query

Quindi vengono create tante schede quante sono le tuple estratte dalla select, ciascuna delle quali riporta il nome della colonna e vicino il valore corrispondente. Il codice illustrato mostra la potenza dello *useState* descritto in precedenza, in quanto la variabile *queryResult* viene utilizzata come contenitore di tutti i tag Ionic che rendono possibile questa interfaccia grafica.

La *Figura 36* illustra quanto appena detto e la funzione ***cardCreator*** che crea dinamicamente tutta la struttura grafica:

```
const [queryResult, setQueryResult] = useState<any>();

function cardCreator(rows: any, attribute: any[]){
  let i=0;
  setQueryResult( rows.map((row : any)=>
    <IonGrid fixed>
    <IonRow>
      <IonCol>
        <IonCard color="secondary">
          <IonCardHeader>
            <IonCardTitle>RESULT Nº {i++} </IonCardTitle>
          </IonCardHeader>
          <IonCardContent>
            {attribute.map( col => <p>{col} : {row[col]}</p>)}
          </IonCardContent>
        </IonCard>
      </IonCol>
    </IonRow>
  </IonGrid>
))
}
```

Figura 36: Codice della funzione *cardCreator*

Ricordando che a questa funzione vengono passati come parametri l'array multidimensionale contenente tutte le tuple estratte dalla query e l'array contenente tutti i nomi delle colonne, è chiaro il perché è presente un *map()* interno ad un *map()* che corrisponde ad un ciclo for nidificato all'interno di un secondo ciclo for. Il primo serve per estrarre una *row* che è un elemento del primo array, il secondo invece serve per iterare all'interno di questa tupla ed estrarne tutti i valori, utilizzando come chiavi i nomi delle colonne. Così facendo viene creato dinamicamente l'insieme delle strutture che vanno a formare le *IonCard* illustrate nella precedente *Figura 35*.

Infine si vuole far notare come l'ultima riga di codice della *Figura 34* corrisponde ad un *catch* corrispondente all'esecuzione della query: se questa va a buon fine si attiva tutto il processo sopra descritto, ma nel caso in cui vi sia un errore (per esempio a causa di un'errata immissione dei dati da parte dell'utente oppure un risultato vuoto) viene attivato il catch che setta a "true" lo stato della variabile *showWrongQueryAlert*. A questa variabile corrisponde una finestra di dialogo che appare nella parte alta dello schermo per dare delle informazioni all'utente e può essere chiusa solo manualmente. Il codice relativo a questo tag Ionic è visibile nella *Figura 37*:

```
<IonAlert
  isOpen={showWrongQueryAlert}
  onDidDismiss={() => setShowWrongQueryAlert(false)}
  cssClass='secondary'
  animated
  header={'Attenzione'}
  message={'Tabella risultante vuota'}
  buttons={['OK']}
/>
```

Figura 37: Codice che produce la finestra di alert

Questa finestra, definita da Ionic per mezzo di un tag Ionic vuoto, può essere modificata solo tramite i suoi attributi. L'attributo *isOpen* può ricevere solo un valore booleano (nel caso sia valorizzato "true" viene mostrata la finestra di alert), *onDidDismiss* indica un insieme di azioni che devono essere fatte nel momento in cui la finestra viene chiusa (in questo caso setta a "false" il valore della variabile *showWrongQueryAlert* per non rendere più visibile la finestra); *header, message* e *buttons* sono gli elementi grafici che verranno visualizzati dall'utente.

5.7 Interfaccia per la sincronizzazione dei database

Non resta che parlare dell'ultima pagina, che risulta la più semplice graficamente ma permette di eseguire il compito centrale di questo sviluppo. La sua parte grafica è mostrata nella seguente *Figura 38*:

Figura 38: Interfaccia grafica della pagina di sincronizzazione

Presenta quindi un unico bottone che nel momento in cui viene premuto attiva la funzione ***syncDb*** che invierà le tuple estratte dal database interno all'applicazione, o meglio solo quelle che sono state inserite o modificate dalla data dell'ultima sincronizzazione in poi, al Web server e riceverà da questo una risposta di tipo JSON.
Inizierà ora la descrizione della funzione appena nominata ma, a causa della sua complessità, verrà scomposta in più sezioni e quindi anche il codice verrà riportato tramite figure multiple.

Nella prima parte ovviamente avviene il collegamento al database interno all'applicazione tramite il metodo, ormai noto, *SQLite.create()*. Vengono poi dichiarate alcune variabili che serviranno successivamente all'interno della funzione, e viene

richiamata la funzione *setShowLoading* che utilizza l'hook *useState* e pone una variabile a true, creando una schermata di caricamento visibile all'utente avvertendolo che la sincronizzazione è in corso. Viene effettuata poi una prima query al database interno che estrae dalla tabella *last_sync* la data e l'ora dell'ultima sincronizzazione e pone la variabile *lastUpdate* uguale al valore appena estratto (*Figura 39*).

```
function syncDb(){
  SQLite.create({ name: 'dbprova.db', location: 'default'})
  .then(db => {
    setShowLoading(true);
    let paramString : string = "";
    let rowsSet : any[] = [];
    let primaryKey : string = "";
    db.executeSql("SELECT * FROM last_sync", [])
    .then(tmp => {
      let lastUpdate : string = tmp.rows.item(0)['last_update'];
```

Figura 39: Codice che estrae data e ora dell'ultima sincronizzazione

Successivamente viene effettuata una seconda query alla base di dati, che estrae le informazioni riguardanti la tabella che si sta aggiornando tramite *"PRAGMA table_info('table')"* e il risultato che si ottiene viene posto all'interno di un ciclo for per estrarne le informazioni. Infatti il risultato che si ottiene da questa query mostra vari dettagli sulla tabella e i suoi attributi: analizzandolo si nota come tutti gli attributi che non sono *primary key* hanno un valore "pk=0" mentre per gli attributi che lo sono, questo viene valorizzato con una numerazione crescente ("pk=1" per la prima chiave, "pk=2" per la seconda etc). Si crea allora una stringa, che diventerà parte dell'URL della richiesta GET, formattata nel seguente modo: "pk=valore1&pk=valore2&pk=valore3". Ovvero per ogni chiave primaria verrà aggiunto un "pk=valore" alla stringa ***primaryKey***, dove al posto di "valore" si immette il nome dell'attributo che è chiave primaria. Nel caso in cui l'esecuzione della query per l'estrazione delle informazioni della tabella dia un errore, non si attiverà il *then* ma il *catch*, che terminerà la schermata di caricamento che l'utente stava visualizzando e apparirà una seconda finestra di alert che gli comunica un'errata estrazione dalla tabella.

Quanto descritto sopra è mostrato nella *Figura 40*:

```
db.executeSql('PRAGMA table_info('+table+')', [])
  .then(data => {
    for(var i=0; i<data.rows.length; i++){
      if(data.rows.item(i).pk!=0){
        primaryKey += "pk="+data.rows.item(i).name+"&"
      }
    }
 })
  .catch(()=> {
    setShowLoading(false);
    alert("Tabella non valida")
  })
```

Figura 40: Codice che estrare le informazioni relative alla tabella da sincronizzare

Bisogna eseguire una terza query per l'estrazione dei dati della tabella. Poiché si devono sincronizzare solo quelle tuple che sono state inserite o aggiornate rispetto all'ultima sincronizzazione, si andranno ad estrarre in base al fatto che il campo "last_update" abbia un valore maggiore rispetto alla variabile *lastUpdate* valorizzata in precedenza (vedi *Figura 39*). Dall'insieme risultante si estraggono i nomi degli attributi, che saranno immessi all'interno dell'array ***paramKeys*** il quale servirà sia nel ciclo for successivo per l'estrazione dei valori dall'insieme dei dati che per la strutturazione dell'URL. Quindi una volta effettuata la query e valorizzato l'array *paramKeys* con i nomi degli attributi, si devono estrarre i dati delle tuple contenuti in un oggetto strutturato e inserirli all'interno di un array di array che andrà a rappresentare l'intero set di tuple estratto dalla tabella. Per fare ciò si utilizzano due cicli for nidificati, il primo che serve per iterare all'interno dell'oggetto contenente i dati e selezionare una tupla, mentre il secondo itera la tupla stessa e ne estrae il contenuto. Per fare questo si utilizzerà un array di appoggio *paramValues* che, ad ogni nuova iterazione del ciclo for esterno, verrà riempito con i valori di una della tupla *i-esima*; al termine del for interno, quindi quando l'array di appoggio conterrà tutti i valori appartenenti ad una determinata tupla, verrà utilizzato il metodo *push* per inserire questo array in coda all'ultimo elemento dell'array ***rowsSet***.

Così facendo al termine dei due cicli for, *rowsSet* conterrà l'intero insieme di tuple da sincronizzare.

Il procedimento descritto è mostrato in *Figura 41*:

```
//Estrae dall'app le rows che sono state aggiornate/inserite dall'ultima sync in poi
db.executeSql("SELECT * FROM "+ table +" WHERE last_update>'"+lastUpdate+"'", [])
  .then(data => {
    // Riempie l'array paramKeys con i nomi delle colonne della tabella
    Object.keys(data.rows.item(0)).map(key=>{
      paramKeys.push(key);
    })
    // Riempie l'array di appoggio paramValues con i valori della row
    //Ad ogni ciclo del primo for, si riempie nuovamente l'array paramValues
    let paramValues : any[] = [];
    for (var i = 0; i < data.rows.length; i++) {
      paramValues = [];

      for(var j = 0; j< paramKeys.length; j++){
        paramValues[j]=data.rows.item(i)[paramKeys[j]];
      }
      //Pusha l'i-esima row(rappresentata da paramValues) nell'array di array
      rowsSet.push(paramValues);

    }
```

Figura 41: Codice che riempie *rowsSet* con i dati delle tuple estratte

Dopo che tutte le tuple da sincronizzare sono state estratte e i valori sono stati messi in delle strutture opportune, non resta che attivare il processo di sincronizzazione. Per ciascuna tupla, quindi per ciascun elemento dell'array *rowsSet*, dovrà essere formattato un URL in modo tale da comunicare i dati della tupla al server che avvierà le logiche di sincronizzazione. Per ciascuna row verrà eseguita una richiesta di tipo GET e verrà utilizzata la libreria Axios, già introdotta alla fine del *capitolo 5.4*; poiché questa libreria utilizza la logica delle *promise* è necessario inserire tutte le promises, ciascuna collegata ad una determinata richiesta GET, all'interno di un array e utilizzare successivamente il metodo *Promise.all()* per aspettare che vengano tutte risolte correttamente.

La *Figura 42* mostra quanto appena descritto:

```
//Array di promise, che conterrà tutte le Axios request
const promises =[];
let finalUrl : string;
//Per ogni row contenuta in rowSet crea un URL e pusha la Axios Request nell'array di promise
for (var i=0; i<rowsSet.length; i++){
  //Ad ogni nuovo ciclo for, vengono rituilizzate le seguenti stringhe
  paramString = "";
  finalUrl="http://192.168.11.246:8080/prova/sync.jsp?";

  {/* Formatta la stringa paramString(nomeParametro=valoreParametro) da aggiungere all'URL */}
  for (var j = 0; j < paramKeys.length; j++) {
    paramString += paramKeys[j]+"="+rowsSet[i][j]+"&";
  }
  finalUrl +=paramString+"table="+table+"&"+primaryKey+"last_sync_update="+lastUpdate;
  console.log(finalUrl);
  console.log("promsie "+i);
  //Fa il push nell'array di promise della richiesta
  promises.push( axios.get(finalUrl)
```

Figura 42: Codice che formatta l'url ed invia la richiesta GET

Se si guarda attentamente l'ultima linea di codice, la parentesi del metodo *push* viene aperta e mai chiusa. Questo poiché la richiesta GET non è l'unico codice relativo a ciascuna *promise*, ma vi è anche il codice che descrive come l'applicazione deve utilizzare la risposta che riceve dal server. Ciò viene scritto all'interno del *then* successivo alla richiesta Axios.

L'applicazione può ricevere una risposta formattata in due differenti maniere: nel caso in cui la tupla inviata per la sincronizzazione è risultata uguale ad una del database esterno, ma in quest'ultimo la riga è stata eliminata, allora si dovrà eliminare quella tupla anche dal database interno all'App. Se invece è stata riscontrata un'uguaglianza dei valori delle chiavi primarie delle due tuple, ciò significa che nel database esterno è presente una tupla con le chiavi ugualmente valorizzate ma che avrà, probabilmente, gli altri campi con valori diversi: in questo caso si aggiorna la tupla della base di dati interna all'App. In entrambi i casi quindi il database esterno, essendo quello "centrale", è stato ritenuto ad importanza maggiore e quindi nel caso di corrispondenza di rows viene aggiornato il database interno dell'applicazione.

L'App riceverà dal server una risposta JSON solo in relazione a quelle tuple da aggiornare nell'App; questa risposta avrà un campo denominato "*operation*" che assumerà i valori

"*delete*" oppure "*update*". Nel primo caso saranno presenti altri campi che indicheranno le chiavi corrispondenti alla tupla da eliminare, mentre nel secondo caso oltre alle chiavi saranno inviati tutti gli attributi con i rispettivi valori, che dovranno essere utilizzati per aggiornare i precedenti valori della tupla.

La *Figura 43* mostra il codice che si occuperà di eseguire l'aggiornamento della base di dati interna all'applicazione:

```
//Stringa formata da un insieme di parametri nomeP=valueP, necessario per l'operazione di update
let queryValues :string ="";
//Variabile che contiene l'operazione da attuare (delete/update)
let operationQuery :string ="";
//Stringa contenente i parametri relativi al WHERE della query
let whereParams : string ="";
//Itera per tutto il JSON
for( var key in response.data){
  //Rowstatus e last_update non ci interessano in quanto relativi alla row del DB esterno
  //e pk è l'array di primaryKey necessario nel WHERE della query
  if(key=="rowstatus" || key=="last_update" || key=="pk") console.log("Cose da non usare");
  //Setta l'operazione da fare
  else if(key=="operation") operationQuery = response.data[key];
  else{
    //Questo else-if serve per la diversa formattazione nel caso in cui il parametro abbia
    //come valore una stringa(e una data) oppure un numero
    if(!isNaN(response.data[key])){
      {/* Is a number */}
      queryValues += key+"="+response.data[key]+",";
   } else queryValues += key+"='"+response.data[key]+"',";
  }
}


//Prende l'array 'pk' passato nel JSON, contenente i NOMI delle primaryKeys, e lo itera con il map
response.data["pk"].map((primKey:any)=>{
  //Tramite la primaryKey selezionata in questa iterazione, riprendiamo il JSON ricevuto come
  //risposta e la usiamo come chiave per recuperare il relativo VALORE della primKey
  //l'if-else serve come prima per l  a diversa formattazione tra stringa e numero
  if(!isNaN(response.data[primKey])){
    {/* Is a number */}
    whereParams += " "+primKey+"="+response.data[primKey]+" and";
 } else whereParams += " "+primKey+"='"+response.data[primKey]+"' and";
})
//Lo slice serve per eliminare l'ultimo and (non necessario) aggiunto nell'ultima iterazione
whereParams=whereParams.slice(0,-3);

//Fa una query diversa nel caso in cui l'operazione richiesta sia quella di delete o update
if(operationQuery=="update"){
  db.executeSql("UPDATE "+table+" SET "+queryValues+"rowstatus='u', last_update=DATETIME('now') WHERE "+whereParams
} else if (operationQuery=="delete"){
  db.executeSql("DELETE FROM "+table+" WHERE "+whereParams);
}
```

Figura 43: Codice che esegue le operazioni di sincronizzazione sul database interno

Nella parte iniziale del codice vengono dichiarate tre variabili di tipo stringa: *queryValues* che verrà utilizzata solo nel caso in cui l'operazione da fare sia di aggiornamento della tupla, poiché conterrà i nomi degli attributi con i relativi valori da aggiornare (la stringa

sarà formattata come “nome=valore” in caso di un numero e “nome=’valore’ ” nel caso di una stringa o una data); *operationQuery* sarà uguale all’operazione richiesta, quindi assumerà i valori “*update*” o “*delete*”. Infine la variabile *whereParams* contiene i parametri che saranno necessari per la query, che sia essa di aggiornamento o di eliminazione, in quanto tramite le chiavi primarie si andrà a definire la tupla sulla quale effettuare l’operazione.

Successivamente è presente un ciclo for che itera l’intero JSON ricevuto come risposta, identificato da “*response.data*”, dal quale sarà necessario estrarre il valore dell’operazione da effettuare (che ha come chiave il temine “*operation*”) e il valore degli attributi (è presente un costrutto *if-else* per determinare se il valore dell’attributo è un numero oppure no e quindi formattare nel modo corretto la stringa).

Dopo aver fatto ciò è necessario estrarre il valore delle chiavi primarie per determinare univocamente la tupla da aggiornare all’interno della tabella. Per questo scopo viene utilizzato l’array “*pk*” inviato nel JSON che contiene tutti i nomi degli attributi che sono chiavi. Utilizzando il metodo *map* viene quindi iterato l’array e formattata la stringa *whereParams* che rappresenterà nomi e valori degli attributi che sono anche chiavi.

Infine non resta che eseguire la query formattata in base all’operazione e questo viene fatto tramite il metodo *executeSql*.

Questa operazione di sincronizzazione viene ripetuta per tutte le tabelle presenti nel database ed una volta terminate le operazioni per ciascuna di esse, viene aggiornato il campo *last_update* della tabella *last_sync* con la data e l’ora del momento di conclusione della sincronizzazione.

Abbiamo così terminato l’analisi del codice che va a costituire l’applicazione sviluppata. Nelle prossime sezioni verrà descritto il codice Java che racchiude tutta la logica lato server e rende la sincronizzazione possibile.

5.8 Creazione della base di dati esterna PostgreSQL

Si passa ora ad una veloce panoramica del database esterno, che utilizza come RDBMS PostgreSQL. In esso sono state create tre tabelle, tramite i comandi illustrati in *Figura 44:*

```
CREATE TABLE IF NOT EXISTS master(order_id INT PRIMARY KEY, client VARCHAR(30),
 client_id INT, order_date DATE, rowstatus CHAR, last_update timestamp without time zone,
 dirty_bit CHAR );


CREATE TABLE IF NOT EXISTS details(order_id INT, order_num INT , article VARCHAR(30),
 article_id INT, long_desc VARCHAR(200), quantity SMALLINT,rowstatus CHAR,
 last_update timestamp without time zone, dirty_bit CHAR, PRIMARY KEY (order_id, order_num),
 FOREIGN KEY (order_id) REFERENCES master(order_id));


CREATE TABLE IF NOT EXISTS client(client_id INT PRIMARY KEY, client_name VARCHAR(30),
 phone VARCHAR(10), location VARCHAR(30));
```

Figura 44: Codice che crea le tabelle nella base di dati

Come esposto nel *capitolo3.3*, questo database risulta composto da tre tabelle nonostante solo due risultino effettivamente da aggiornare, per verificare che non vi siano errate codifiche nelle logiche di sincronizzazione.

Sono stati anche creati dei trigger che permetteranno di modificare i campi "*rowstatus*" e "*last_update*" delle tabelle *master* e *details*. Questo poiché i due attributi non devono essere inseriti manualmente dall'utente, essendo essi necessari per la corretta sincronizzazione, e quindi deve essere il database ad occuparsi di inserire o aggiornare i valori di questi campi.

La *Figura 45* mostra le funzioni che verranno utilizzate dai trigger per svolgere questo compito; queste funzioni possono essere utilizzate per qualsiasi trigger, ma questi ultimi saranno invece specifici per ogni tabella.

```
CREATE FUNCTION add_insert_trigger()
RETURNS trigger AS '
BEGIN
     IF NEW.rowstatus IS NULL THEN
       NEW.rowstatus := ''i'';
     END IF;
        IF NEW.last_update IS NULL THEN
       NEW.last_update := now()::timestamp;
     END IF;
     RETURN NEW;
END' LANGUAGE 'plpgsql';
```

```
CREATE FUNCTION update_trigger_function()
RETURNS TRIGGER AS $BODY$
   BEGIN
      NEW.last_update := now();
        NEW.rowstatus = 'u';
RETURN NEW;
   END $BODY$ LANGUAGE plpgsql;
```

Figura 45: Funzioni utilizzate dai trigger

La prima funzione *add_insert_trigger* verrà utilizzata dai trigger relativi all'inserimento di nuove tuple e, avendo queste i campi "rowstatus" e "last_update" possibilmente vuoti poiché non inseriti dall'utente, li valorizzeranno.

La funzione *update_trigger_function* verrà utilizzata per l'aggiornamento di una tupla: in questo caso bisogna aggiornare il campo "rowstatus" ponendolo ad "*u*" e il campo "last_update" con la data e l'ora dell'aggiornamento.

Si creerà poi il trigger relativo ad una tabella utilizzando questa la sintassi mostrata in *Figura 46*:

```
CREATE TRIGGER update_trigger

BEFORE UPDATE ON master

FOR EACH ROW

EXECUTE PROCEDURE update_trigger_function();
```

Figura 46: Codice che crea il trigger per la tabella Master

Nello specifico questo codice crea un trigger denominato *update_trigger* relativo alla tabella *master* e, per ciascuna tupla coinvolta nel processo di aggiornamento, esegue la procedura *update_trigger_function* sopra descritta.

Ripetendo la creazione di trigger per le funzioni e tabelle necessarie, e inserendo dei dati all'interno delle stesse, si completa la realizzazione del database esterno.

5.9 Web Server e JSP

Dopo aver visto le differenti componenti dell'applicazione e il database esterno, non resta che trattare i moduli presenti nel Web server che chiudono il ciclo della logica di sincronizzazione.

Come si è visto nella *Figura 28* e *Figura 42*, l'App effettua una richiesta GET al server, ma richiedendo due pagine differenti: la prima, necessaria per l'inizializzazione del database interno estraendo i dati dal database esterno, è *dbGetData.jsp* mentre la seconda, che attua la sincronizzazione tra le basi di dati, è *sync.jsp* . Si dovrà quindi analizzare il codice di entrambe le JavaServer Pages e anche in questo caso il codice verrà separato in diverse sezioni per una migliore comprensione.

5.9.1 dbGetData.jsp

La tecnologia delle JavaServer Pages permette di utilizzare codice Java misto ad HTML e JavaScript, per questo motivo è necessaria una divisione tra i differenti tipi di codice. Questo si ottiene racchiudendo il codice Java all'interno dei seguenti tag di apertura "<%@" e di chiusura "%>".

Nella parte iniziale del codice, come in un normale file Java, devono essere importate le librerie che verranno utilizzate, che in questo caso sono mostrate nella *Figura 47*:

```
<%@ page import="java.sql.*" %>
<%@ page import="java.io.*" %>
<%@ page import="com.google.gson.*"%>
<%@ page import="java.util.*" %>
<%@ page import="org.json.*" %>
```

Figura 47: Librerie importate all'interno della JSP

Successivamente inizia l'elaborazione della richiesta: *request* è l'oggetto che fornisce le informazioni sulla richiesta effettuata dal client ad un servlet; il servlet container crea un oggetto *ServletRequest* (che viene ereditato da *request*) il quale ha definito un set di

metodi, tra i quali *getParameter*. Quest'ultimo ritorna il valore di un parametro della richiesta come una stringa, oppure *null* nel caso in cui il parametro non esiste.
Nel caso di questo sviluppo il parametro che viene passato all'interno della richiesta è *table* e il suo valore verrà salvato nell'omonima variabile di Java, come illustrato in *Figura 48*:

```
String table = request.getParameter("table");
```

Figura 48: Codice che estrae il valore del parametro della richiesta

Avendo acquisito l'informazione riguardante la tabella dalla quale bisogna estrarre i dati, si può procedere con il collegamento al database esterno per poter effettuare la query. Si utilizzerà il JDBC che tramite una API ad oggetti permette di relazionarsi facilmente alla base di dati. Prima di tutto bisogna caricare i driver necessari per il collegamento, i quali variano con il database utilizzato. Successivamente si può utilizzare il metodo *getConnection* del *DriverManager* per stabilire la connessione e questa viene salvata nell'oggetto di tipo Connection *dbConnection* (tramite il quale sarà possibile anche chiudere la connessione una volta terminate le operazioni). Per connettersi al database è necessario specificare l'URL (che contiene il nome del database), l'utente e infine la password dell'RDBMS, all'interno del metodo *getConnection*. Prima di procedere ad effettuare la query, bisogna creare un oggetto che rappresenti l'SQL statement, il quale genera anche l'oggetto *ResultSet* che è una tabella di dati che rappresenterà i dati estratti dalla query. Creato anche lo statement tramite il metodo *createStatement* collegato alla connessione precedentemente stabilita, non resta che utilizzare il metodo *executeQuery* che permette di eseguire la query. Quanto descritto è mostrato in *Figura 49*:

```
// Load JBBC driver "org.postgresql.Driver"
Class.forName("org.postgresql.Driver").newInstance();

// (URL, user, password)
Connection dbConnection = DriverManager.getConnection(
                    "jdbc:postgresql://localhost:5432/dbprova",
                    "postgres",
                    "password");
Statement getFromDb = dbConnection.createStatement();
ResultSet queryResult  = getFromDb.executeQuery("SELECT * FROM "
                                    +table+"WHERE rowstatus <>'d'");
```

Figura 49: Codice che connette la JSP con il database esterno

Il risultato della query viene salvato all'interno della variabile *queryResult*. Successivamente vengono dichiarate delle variabili: *metaData* estrae i metadati del risultato della query tramite il metodo *getMetaData*, *numberOfColumns* estrae dai metadati il numero delle colonne tramite il metodo *getColumnCount*, *nameOfColumns* è l'array che conterrà i nomi delle colonne della tabella e ha una grandezza pari a *numberOfColumns*. Infine vengono dichiarati una variabile di tipo JSON *tmpJson*, che rappresenterà una row diversa ad ogni ciclo, e *response* che è di tipo array di JSON e conterrà i dati di tutte le tuple e quindi l'intera tabella. La *Figura 50* mostra le dichiarazioni delle variabili:

```
ResultSetMetaData metaData = queryResult.getMetaData();
int numberOfColumns = metaData.getColumnCount();
//Array formato dai nomi delle colonne (da riempire)
String[] nameOfColumns = new String[numberOfColumns];
//response è larray dove verranno inseriti tutti i json
JSONArray response = new JSONArray();
//tmpJson è il json che ogni volta rappresenterà una nuova
//row che sarà inserita nel JSON-array response
JSONObject tmpJson = new JSONObject();
//Il flag serve per estrarre i nomi delle colonne solo per
//la prima iterazione
boolean flag = true;
```

Figura 50: Dichiarazioni di variabili

A seguire viene eseguito un ciclo while che scorrerà i risultati della query tramite il metono *next*; *queryResult* è infatti un oggetto di tipo *resultSet*, che mantiene un cursore che punta ad una determinata riga di dati ed è inizialmente posizionato prima della riga

iniziale, e restituisce *false* quando il cursore si muove oltre l'ultima riga. Ad ogni iterazione quindi *queryResult* rappresenta una determinata tupla della tabella.

Solo durante la prima iterazione vengono estratti i nomi delle colonne, utilizzando un ciclo for che itera l'oggetto contenente i metadati, che vengono memorizzati all'interno dell'array *nameOfColumns*.

Per ogni iterazione invece viene istanziato nuovamente il *tmpJson*, che verrà poi riempito per mezzo di un ciclo for con tante coppie "nome:valore" corrispondenti al nome della colonna e il valore di quell'attributo per una data tupla. Il JSON così formato verrà poi messo all'interno dell'array di JSON *response* e si passerà all'iterazione successiva.

Quando viene iterato l'intero *queryResult* e quindi *response* contiene i dati dell'intera tabella estratta, l'array di JSON viene stampato a video in modo tale da essere utilizzato dall'applicazione. La *Figura 51* rappresenta il codice appena descritto:

```
    while(queryResult.next()){
        if(flag){
            try{
                //Questo for estrae il nome delle colonne
                for(int i=1;i<=numberOfColumns;i++){
                    String columnName = metaData.getColumnName(i);
                    nameOfColumns[i-1] = columnName;
                }
                flag=false;
            } catch(Exception e){
                out.print(e);
            }
        }
        tmpJson = new JSONObject();
        //Questo for estrae il valore delle celle di una riga
        //Faccio iniziare da 1 poichè gli indici delle righe iniziano da 1
        for(int i=1;i<=numberOfColumns;i++){
            if(nameOfColumns[i-1]!="dirty_bit"){
                tmpJson.put(nameOfColumns[i-1], queryResult.getString(i));
            }
        }

        response.put(tmpJson);
    }
dbConnection.close();
out.print(response.toString());
%>
```

Figura 51: Creazione della risposta JSON

5.9.2 sync.jsp

Questo secondo file è ben più articolato del precedente, in quanto deve mettere in atto una logica abbastanza complessa. Per questo motivo è stato realizzato lo schema di *Figura 52* che mostra in maniera sequenziale quelli che sono i vari step che verranno affrontati nell'esecuzione del codice:

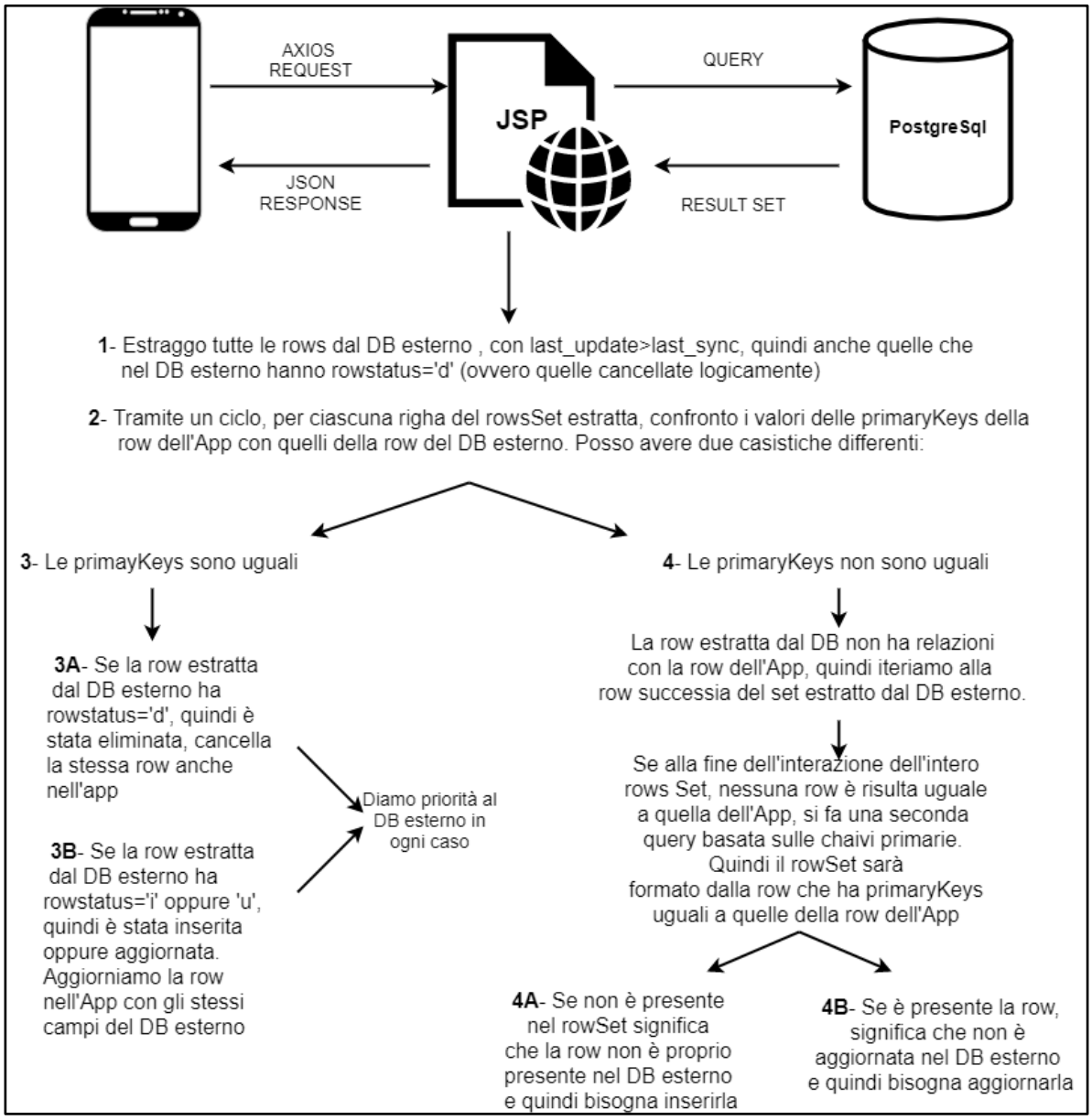

Figura 52: Schematizzazione della logica di sincronizzazione

Il codice è quindi articolato in quattro step principali, con gli ultimi due (3 e 4) che hanno delle sotto-sezioni. Anche il codice della JSP risulta indentato e diviso, tramite commenti, nelle stesse aree per una maggiore comprensione; data la lunghezza del codice non

verranno riportati gli screen di tutto il codice ma solo delle procedure ritenute più importanti.

Come nel file precedente vengono presi tutti i parametri passati nell'URL, ma in questo caso, essendo molteplici, non si utilizzerà il *getParameter* (che si deve usare nel caso di parametro singolo) ma il metodo *getParameterMap* che restituisce un oggetto di tipo *Map*, ovvero mappa le chiavi con i rispettivi valori. In questo caso le chiavi sono i nomi dei parametri, mentre i valori sono appunto i valori assunti dai parametri e questi vengono tutti considerati come array (quindi avremo array costituiti da più elementi solo nel caso in cui un parametro venga ripetuto più volte, come accade per le primary keys che utilizzano tutte il parametro "*pk*"). Successivamente si vanno ad estrarre i valori di questi parametri, al momento contenuti nella variabile *parameters*, e si valorizzano altre variabili tramite un ciclo for che lo itera: la variabile *table* conterrà il nome della tabella che si vuole sincronizzare e verrà utilizzato per la query di estrazione dei dati dal database esterno, *lastSync* conterrà la data e l'ora dell'ultima sincronizzazione e anche questo verrà utilizzato per la query, *primaryKey* è l'array che conterrà tutti i nomi dei campi che sono chiavi primarie della tabella, infine *parametersJson* è un oggetto JSON e conterrà tutti gli altri parametri (ovvero le coppie "campo:valore" che dovranno essere confrontati con quelli estratti dal database).

In seguito, come mostrato nel precedente file JSP, si stabilisce una connessione con il database esterno tramite JDBC e si estraggono, tramite una select query, tutte le tuple che sono state inserite/aggiornate/eliminate (si ricordi che nel database esterno viene attuato un delete logico delle tuple) dall'ultima sincronizzazione.

Si ottiene così un *resultSet* che è possibile interare tramite un ciclo while e l'utilizzo del metodo *next*; durante la prima iterazione vengono anche estratti i nomi dei campi e inseriti all'interno dell'array *nameOfColumns*. Si riempie poi un oggetto JSON, *tmpJson*, con i valori della tupla attualmente puntata nel *resultSet*.

Ora che si hanno sia i dati della tupla inviata dall'App, nella variabile *parametersJson*, e quelli relativi alla tupla del *resultSet*, contenuti dalla variabile *tmpJson*, si possono

confrontare questi oggetti JSON (punto 2 della *Figura 52*): per farlo è necessario fare un confronto sui valori delle chiavi primarie e se anche solo uno di essi risulta differente le tuple non sono uguali. Il codice che realizza il confronto delle variabili è mostrato in *Figura 53*:

```
for ( int i = 0; i< primaryKey.size(); i++){
    jsonToCompare = parametersJson.getString(primaryKey.get(i));
    if(!(jsonToCompare.equals(tmpJson.getString(primaryKey.get(i))))){
        isRowEqual = false;
        //serve per uscire immediatamente dal ciclo
        i = primaryKey.size();
    }
}
```

Figura 53: Codice che confronta le chiavi delle tuple per vedere se sono uguali

Nel caso in cui le tuple risultino effettivamente uguali, la variabile *isRowEqual* rimarrà a "true" e si entrerà all'interno di un blocco *if* (punto 3 della *Figura 52*). Si controlla allora il valore del campo "rowstatus" della tupla del database esterno: se questa risulta pari a "d" (ha quindi ricevuto un'eliminazione logica nella base di dati esterna) si strutturerà una risposta JSON che indicherà all'applicazione di eliminare fisicamente quella tupla nel suo database interno; in ogni altro caso, quando "rowstatus" assume il valore "i" oppure "u", viene creato un JSON che permetterà all'App di aggiornare i valori della tupla. La prima opzione (punto 3A della *Figura 52*), relativa all'operazione di cancellazione, indicherà nel JSON un campo "*operation*" con valore "*delete*" più i campi relativi alle chiavi primarie e i loro valori. Il secondo caso (punto 3B della *Figura 52*) strutturerà in modo simile il JSON, ma con valore del campo "*operation*" pari ad "*update*" e aggiungendo anche i valori dei campi della tupla.

Il codice relativo a quanto descritto è mostrato in *Figura 54*:

```
if(isRowEqual){
    JSONObject responseToApp = new JSONObject();
    if(tmpJson.getString("rowstatus").equals("d")){
        responseToApp.put("operation","delete");
        for( int i = 0; i< primaryKey.size(); i++){
            responseToApp.put(primaryKey.get(i),parametersJson.getString(primaryKey.get(i)));
        }
        responseToApp.put("pk",primaryKey);
        out.print(responseToApp);
    }
    else{
        responseToApp.put("operation","update");
        for( int i=0; i<numberOfColumns;i++){
            responseToApp.put(nameOfColumns[i],tmpJson.getString(nameOfColumns[i]));
        }
        out.print(responseToApp);
    }
    //Questo while itera fino alla fine delle row estratte dal db esterno,
    //in quanto la row uguale è già stata trovata
    while(queryResult.next()){}
}
```

Figura 54: Codice che crea il JSON per l'aggiornamento del database dell'App

Se però nessuna tupla estratta in base alla data di ultima sincronizzazione risulta uguale a quella dell'App, bisogna fare una nuova query al database esterno e vedere se è presente una tupla con gli stessi valori delle chiavi primarie (punto 4 della *Figura 52*). La seguente *Figura 55* mostra in che modo viene strutturata la query:

```
if(!isRowEqual){
   //Crea la query dove richiede le row con determinate pk
   String newQuery = "SELECT * FROM "+table+" WHERE";
   for( int i = 0; i< primaryKey.size(); i++){
       int isNumber;
       try {
          isNumber = Integer.parseInt(parametersJson.getString(primaryKey.get(i)));
          if(i == (primaryKey.size()-1) ){
              newQuery += " "+primaryKey.get(i)+"="+parametersJson.getString(primaryKey.get(i));
          }
          else newQuery += " "+primaryKey.get(i)+"="+parametersJson.getString(primaryKey.get(i))+" and";
       }
       catch (NumberFormatException e)
       {
           if(i == (primaryKey.size()-1) ){
               newQuery += " "+primaryKey.get(i)+"= '"+parametersJson.getString(primaryKey.get(i))+"'";
           }
           else newQuery += " "+primaryKey.get(i)+"= '"+parametersJson.getString(primaryKey.get(i))+"' and";
       }
   }
   ResultSet equalPrimaryKeySet = getFromDb.executeQuery(newQuery);
```

Figura 55: Codice che struttura la query con le chiavi primarie

Il costrutto *try-catch* è un espediente utilizzato per controllare se il valore di una chiave è un numero oppure no: nel caso in cui non lo sia, viene generata un'eccezione e si esegue il codice del blocco *catch*, che racchiude il valore tra virgolette.

Successivamente a questo blocco di codice, si controlla se il *resultSet* della query è vuoto e quindi non c'è nessuna tupla con le chiavi primarie valorizzate ugualmente alla tupla dell'App, oppure è presente un elemento. Nel primo caso (punto 4A della *Figura 52*) si inserirà la tupla all'interno del database esterno, nel secondo (punto 4B) si aggiorneranno i campi della tupla nella base di dati esterna. I codici di questi due ultimi punti non verranno mostrati in quanto simili ad altri codici visti in precedenza.

Con quest'ultima procedura termina la trattazione delle logiche di sincronizzazione che sono state codificate. Manca però una parte che completa veramente la sincronizzazione dei database: la procedura fino ad ora descritta infatti, attua una sincronizzazione in base alle tuple inviate dall'App, ma le righe che sono state inserite o aggiornate nel database esterno e non sono uguali a nessuna tupla inserita o aggiornata dell'App non vengono considerate e quindi non vengono sincronizzate. Questa omissione non è dovuta ad una mancanza di attenzione dello sviluppatore o ad un'elevata difficoltà nell'attuazione, ma causata dal tempo limitato del tirocinio. La logica che sta dietro questa seconda parte è stata difatti progettata ma non codificata: le tabelle del database esterno hanno un campo in più rispetto a quelle della base di dati dell'App (come già esposto nel *capitolo 3.3*) denominato ***dirty_bit*** (in conformità agli algoritmi riguardanti la memoria virtuale negli elaboratori) che può assumere solo due valori "f" o "t". Quando una tupla viene inserita o aggiornata, questo campo viene posto a "f"; viene cambiato a "t" nel momento in cui la row viene estratta in fase di sincronizzazione e fa quindi parte del *rowsSet* da confrontare. Se questa risulta uguale alla tupla dell'App si eseguono le logiche descritte in questo capitolo e infine si pone nuovamente il campo del *dirty_bit* uguale a "f". Quando queste procedure di sincronizzazione tra App e database esterno saranno concluse, rimarrà impostato a "t" il *dirty_bit* di tutte quelle tuple che sono state estratte dal database esterno ma non sono state sincronizzate con l'applicazione. Deve essere quindi eseguita una

query per estrarre tutte queste tuple (ovvero una select con "*WHERE dirty_bit='t'* ") che saranno inviate come risposta JSON all'App; infine sarà impostato nuovamente a "f" questo campo. Così facendo si ottiene una sincronizzazione completa e finita delle basi di dati.

Capitolo 6

Conclusione e Sviluppi futuri

Come si è potuto notare nei capitoli precedenti, ed in particolare in quello riferito all'applicazione sviluppata (*capitolo 5*), questo progetto ha visto il coinvolgimento di numerose tecnologie. Non tutte sono state affrontate durante il corso di laurea, come l'utilizzo del framework Ionic e React o si pensi allo sviluppo di un'applicazione con l'emulazione di un dispositivo, ma sono state facilmente integrate con quanto appreso durante il corso di Tecnologie Web. Inoltre i linguaggi appresi durante questo corso sono stati visti in una veste leggermente diversa rispetto allo sviluppo classico lato Web client, in quanto utilizzati per la creazione di un'applicazione.

Il progetto portato avanti, inoltre, non aveva nessuna base già strutturata ma è stato costruito da zero in tutte le sue componenti. Lo sviluppo ha portato quindi un graduale e costante aumento delle competenze in tutti questi ambiti; oltretutto nel corso della realizzazione sono state codificate ulteriori funzioni che hanno permesso un potenziamento delle abilità, ma sono state omesse in quanto non ritenute necessarie per gli scopi dell'applicazione. Si fa notare inoltre come anche la logica di sincronizzazione sia stata teorizzata tramite una profonda analisi dei processi necessari per l'attuazione della stessa e non ripresa da schemi pre-esistenti.

Grazie alla scelta di affrontare un tirocinio associato alla tesi, è stato possibile vivere un'esperienza vicina al mondo del lavoro, avendo la possibilità di interagire ed imparare da persone con un bagaglio culturale importante. Oltretutto questo ha permesso di affrontare molti temi importanti del mondo IT: linguaggi Web, sviluppo di applicazioni, creazione ed interfacciamento a database, creazione server e sviluppo in Java lato server.

L’obiettivo del progetto è stato raggiunto pienamente, in quanto si è sviluppato un modulo software per una Web App che riuscisse ad interfacciarsi e sincronizzarsi con un database esterno facendo richieste ad un Web server. La soluzione proposta risulta quindi un’ottima base di partenza che però deve essere ampliata con varie componenti prima di poterla definire un’applicazione finita. Se infatti si pensa all’applicazione MYWO presentata nel *capitolo 2.3.2*, essa si compone di una schermata di login per l’utente, un’interfaccia che presenta in maniera strutturata gli ordini e le loro caratteristiche, oltre ad avere un database centralizzato più ampio e complesso.

Infine si vuole evidenziare come l’azienda presso la quale è stato svolto il tirocinio, alla luce del lavoro eseguito, ha deciso di inserire nei propri archivi il progetto sviluppato, che potrà essere utilizzato sia per sviluppi futuri che come integrazione ad applicazioni che però utilizzano logiche di sincronizzazione legate a software proprietari.

Bibliografia

[1] https://www.smeup.com/blog/blog-software-gestionali-erp/sistemi-gestionali-erp-differenze/ - consultato a Novembre 2020

[2] https://news.microsoft.com/it-it/2013/11/27/2013-16/ - consultato a Novembre 2020

[3] https://www.typescriptlang.org/ - consultato a Novembre 2020

[4] https://nodejs.org/it/ - consultato a Novembre 2020

[5] https://ionicframework.com/docs/cli/commands/serve - consultato a Ottobre 2020

[6] https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements - consultato a Novembre 2020

[7] Atzeni, Ceri, Fraternali – "Basi di dati" - McGraw-Hill Education - 2018

[8] https://www.sqlite.org/index.html - consultato a Novembre 2020

[9] https://www.postgresql.org/docs/9.2/app-psql.html - consultato a Novembre 2020

[10] https://developer.mozilla.org/it/docs/Web/HTTP/CORS - consultato a Novembre 2020

[11] Jason Hunter – "Java Servlet Programming" – O'Reilly – 2001

[12] Hans Bergsten – "JavaServer Pages" – O'Reilly – 2000

[13] https://it.reactjs.org/docs/react-dom.html#render – consultato a Novembre 2020

[14] https://ionicframework.com/docs/api/router - consultato a Novembre 2020

[15] https://ionicframework.com/docs/api/slides - consultato a Novembre 2020

[16]https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Promise - consultato a Novembre 2020

[17] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch - consultato a Novembre 2020