



# UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

DIPARTIMENTO DI INFORMATICA, SISTEMISTICA E  
COMUNICAZIONE

Laurea Magistrale in Teoria e Tecnologia della Comunicazione

Classe LM-92

## IMPLEMENTAZIONE DI COMPONENTI DI INTERFACCIA UTENTE PER LA WEB APP OMNIA

**Relatore:**

Chiar.mo Prof. Federico Antonio  
Niccolò Amadeo CABITZA

**Candidato:**

Umberto PASINETTI  
Matr. n. 873604

**Correlatore:**

Dott.ssa Frida MILELLA



*Alla mia famiglia  
e a coloro che mi sono stati accanto*



# Indice

<b>Elenco delle figure</b>	<b>ix</b>
<b>Acronimi</b>	<b>xi</b>
<b>Glossario</b>	<b>xiii</b>
<b>Introduzione</b>	<b>xv</b>
<b>1 Inquadramento contestuale</b>	<b>1</b>
1.1 Telemedicina . . . . .	1
1.2 Link-Up e Cerba HealthCare . . . . .	2
1.3 OMNIA . . . . .	3
<b>2 Applicazioni web</b>	<b>5</b>
2.1 Principi generali . . . . .	5
2.2 Cenni storici . . . . .	6
2.3 Architettura . . . . .	8
2.3.1 Livelli . . . . .	8
2.3.2 Tipologie . . . . .	9
<b>3 Soluzioni organizzative</b>	<b>13</b>
3.1 <i>Smart working</i> . . . . .	13
3.2 Metodologia agile . . . . .	14
3.3 <i>Framework Scrum</i> . . . . .	15
3.3.1 Protagonisti . . . . .	16
3.3.2 Eventi . . . . .	17
3.3.3 Strumenti . . . . .	18
<b>4 Soluzioni tecnologiche</b>	<b>21</b>
4.1 React . . . . .	21
4.1.1 JSX . . . . .	23
4.1.2 Componenti . . . . .	26
4.1.2.1 <i>Class vs Function</i> . . . . .	27

4.1.2.2	props . . . . .	29
4.1.2.3	Stato . . . . .	32
4.1.3	<i>Hooks</i> . . . . .	33
4.1.3.1	<i>Built-in hooks</i> . . . . .	34
4.1.3.2	<i>Hooks</i> personalizzati . . . . .	37
4.2	Material UI . . . . .	39
4.2.1	Material Design . . . . .	40
4.2.2	Componenti . . . . .	41
4.3	Altre librerie . . . . .	45
4.3.1	Dialogo client-server . . . . .	45
4.3.1.1	API REST . . . . .	45
4.3.1.2	Axios . . . . .	47
4.3.2	<i>React Router</i> . . . . .	48
4.3.3	<i>Yup</i> . . . . .	49
4.3.4	<i>date-fns</i> . . . . .	50
4.3.5	<i>mui-rte</i> . . . . .	50
<b>5</b>	<b><i>Feature</i></b> . . . . .	<b>53</b>
5.1	Flusso di sviluppo standard . . . . .	54
5.2	F1 - F3: inserimento campo . . . . .	56
5.2.1	F1: visibilità nel sito web . . . . .	56
5.2.2	F2: e-mail . . . . .	60
5.2.3	F3: link referti e portale prenotazione . . . . .	60
5.3	F4: modifica tabella contratti di locazione . . . . .	64
5.4	F5: inserimento campo ricerca . . . . .	76
5.5	F6 - F10: inserimento sezione . . . . .	81
5.5.1	F6: “Magazine” . . . . .	81
5.5.2	F7: “FAQ” . . . . .	105
5.5.3	F8: “Business Unit” . . . . .	105
5.5.4	F9: “Specialistiche” . . . . .	108
5.5.5	F10: “Posizioni aperte” . . . . .	109
<b>Conclusioni</b>		<b>113</b>





# Elenco delle figure

5.1	Modale aggiunta nuova struttura . . . . .	61
5.2	Modale modifica struttura esistente . . . . .	61
5.3	Tabella informazioni struttura . . . . .	62
5.4	Modale aggiunta nuova BU . . . . .	62
5.5	Modale modifica BU esistente . . . . .	62
5.6	Tabella BU esistenti e tabella informazioni BU . . . . .	63
5.7	Modale aggiunta nuovo contratto . . . . .	74
5.8	Modale archiviazione contratto esistente . . . . .	74
5.9	Tabella contratti non espansa . . . . .	74
5.10	Tabella contratti con riga espansa . . . . .	75
5.11	Tabella REO campo ricerca vuoto . . . . .	80
5.12	Tabella REO campo ricerca riempito . . . . .	80
5.13	Modale aggiunta nuovo magazine . . . . .	103
5.14	Modale eliminazione magazine esistente . . . . .	104
5.15	Tabella magazine campo ricerca vuoto . . . . .	104
5.16	Tabella magazine campo ricerca riempito . . . . .	104
5.17	Tabella BU campo ricerca vuoto . . . . .	106
5.18	Tabella BU campo ricerca riempito . . . . .	106
5.19	Pagina dettaglio BU . . . . .	106
5.20	Modale modifica descrizione BU . . . . .	107
5.21	Modale aggiunta immagine BU . . . . .	107
5.22	Modale aggiunta icona BU . . . . .	107
5.23	Modale eliminazione immagine BU . . . . .	108
5.24	Modale eliminazione icona BU . . . . .	108
5.25	Tabella posizioni aperte campo ricerca vuoto . . . . .	109
5.26	Tabella posizioni aperte campo ricerca riempito . . . . .	110
5.27	Modale aggiunta posizione aperta . . . . .	110
5.28	Modale eliminazione posizione aperta . . . . .	110
5.29	Pagina dettaglio posizione aperta . . . . .	111
5.30	Modale modifica posizione aperta . . . . .	111



# Acronimi

**Flash** *Adobe Flash Player.* 7, *vedi Flash*

**AJAX** *Asynchronous JavaScript and XML.* 7, *vedi AJAX*

**API** *Application Programming Interface.* 37, 45–47, 54, 55, 57, 64, 65, 77, 83, 84, 88, 97

**ASD** *Agile Software Development.* 14

**CERN** *Conseil Européen pour la Recherche Nucléaire.* 6, 7

**CRUD** *Create Read Update Delete.* 46

**CSS** *Cascading Style Sheets.* 7, 9, 44, *vedi CSS*

**DB** *DataBase.* 9

**DOM** *Document Object Model.* 22–24, 35, *vedi DOM*

**DRY** *Don't Repeat Yourself.* 27

**ES6** *ECMAScript 6.* 28, *vedi ES6*

**GPS** *Global Positioning System.* 10

**HTML** *HyperText Markup Language.* 6, 7, 9, 24–26, 29, 42, *vedi HTML*

**HTTP** *HyperText Transfer Protocol.* 6, 10, 46, 47, *vedi HTTP*

**IA** *Artificial Intelligence.* 7

**ICT** *Information and Communication Technologies.* 2

**IoT** *Internet of Things.* 2, *vedi IoT*

**IT** *Information Technologies.* 14

**IWA** *Isomorphic Web Application.* 11

**JSX** *JavaScript Syntax Extension.* 23–26, 28, 32

**MPA** *Multiple-Page Application.* 9–11

**MUI** Material UI. x, 3, 21, 39–41, 113

**PWA** *Progressive Web Application.* 10, 11

**QR** *Quick Response.* 11

**REST** *Respresentational State Transfer.* 45, 46

**SEO** *Search Engine Optimization.* 10, 22, *vedi* SEO

**SOAP** *Single Object Access Protocol.* 46

**SPA** *Single-Page Application.* ix, 3, 10, 11, 48

**SSN** Servizio Sanitario Nazionale. 1

**SSR** *Server-Side Rendering.* 9

**UI** *User Interface.* 8, 21, 22, 26, 39, 40, 45, 54, 55, 58, 59, 66, 70, 87, 93, 94, 98

**URL** *Uniform Resource Locator.* 48, 49, *vedi* URL

**UX** *User Experience.* 5, 8, *vedi* UX

**WWW** *World Wide Web.* 6

# Glossario

**AJAX** L'*Asynchronous JavaScript and XML* (AJAX) è un’insieme di tecnologie di sviluppo *software* create per una migliore gestione dell’interattività delle pagine web dinamiche. AJAX consente un leggero disallineamento fra il momento in cui la richiesta viene inoltrata e quello in cui viene restituito il risultato. Il suo obiettivo è quello di far dialogare client e server in *background* per aggiornare il contenuto della pagina senza dover eseguire un *refresh* manuale. 7

**CSS** Il *Cascading Style Sheets* (CSS) è un linguaggio che consente la gestione della veste grafica delle pagine web HTML. Nel 2011 è stata rilasciato il CSS3, la sua attuale versione disponibile. 7

**DOM** Il *Document Object Model* (DOM) rappresenta la struttura di una pagina web tramite una struttura ad albero. Ogni ramo dell’albero termina con un nodo e ogni nodo contiene oggetti. Il DOM collega le pagine web agli script o ai linguaggi di programmazione. 22

**ES6** L'*ECMAScript 6* (ES6) (anche noto come “*ECMAScript 2015*”) è la sesta versione di uno standard per i linguaggi di scripting atto al garantire il corretto funzionamento delle pagine web sui diversi tipi di browser. Con ES6, nel 2015 JavaScript è stato profondamente aggiornato tramite l’introduzione di nuove funzionalità (ad esempio, grazie alle *keyword* *let* e *const*, alla *arrow function*, etc...). 28

**Flash** *Adobe Flash Player* è stato un *software* che per molto tempo ha permesso alle pagine web di essere dinamiche e interattive, ad esempio, riproducendo video. A causa del suo peso e dei problemi di sicurezza a cui esponeva, *Flash* è stato nel corso degli anni lentamente sostituito da tecnologie più leggere e sicure (ad esempio, HTML5) fino a essere abbandonato completamente nel 2020. 7

**HTML** L'*HyperText Markup Language* (HTML) è un linguaggio di *markup* che consente la gestione della struttura di base delle pagine web, ossia l’impaginazione e la formattazione di documenti ipertestuali. Poiché non fornisce vere e proprie istruzioni, non è considerato un linguaggio di programmazione. Nel 2014 è stata rilasciato l’HTML5, la sua attuale versione disponibile. 6

**HTTP** L'*HyperText Transfer Protocol* (HTTP) è un protocollo di livello applicativo usato per la trasmissione di informazioni nel web. Esso opera all'interno di un'architettura client-server. 6

**IoT** L'*Internet of Things* (IoT), in italiano “internet delle cose”, è un paradigma tecnologico che ha come obiettivo la connessione di vari oggetti della nostra quotidianità a internet. Sfruttando la rete, i dispositivi riescono a trasmettere dati provenienti dai sensori di cui sono dotati. 2

**SEO** La *Search Engine Optimization* (SEO) è un insieme di pratiche volte a migliorare la scansione, l'indicizzazione e il posizionamento di informazioni e contenuti di un sito web. Ottimizzare la SEO permette di migliorarne il posizionamento delle pagine nei risultati del motore di ricerca. 10

**URL** L'*Uniform Resource Locator* (URL) rappresenta l'indirizzo di una risorsa (ad esempio, una pagina) all'interno del web; essa è immagazzinata all'interno di un server web, e vi si può accedere, lato client, tramite il browser. 48

**UX** L'*User Experience* (UX), in italiano “esperienza utente”, è il rapporto che si instaura in seguito all'interazione di un individuo con lo strumento in uso, sia esso un prodotto o un servizio. All'interno dell'ampia gamma di fattori che contribuiscono alla UX, alcuni fra i più importanti sono l'utilità, la semplicità d'uso e la piacevolezza. La UI fa parte dell'esperienza utente. 5

**open-source** Il termine *open-source* viene utilizzato per riferirsi a *software* liberamente manipolabili dagli utenti in quanto privi di *copyright*. 21

**responsività** La responsività, in inglese *responsiveness*, è la capacità di un prodotto di adattarsi a diversi contesti d'utilizzo. Il termine viene usato per riferirsi ad applicazioni che, implementando un design *responsive*, sono in grado di gestire la visualizzazione dei contenuti a seconda del dispositivo in uso. 8, 10

**usabilità** L'usabilità, in inglese *usability*, viene definita dalla norma ISO come il «grado in cui un prodotto può essere usato da particolari utenti per raggiungere certi obiettivi con efficacia, efficienza e soddisfazione in uno specifico contesto d'uso». 8

# Introduzione

Ad oggi, in vari paesi del mondo fra cui l’Italia, i tempi sembrano maturi per un importante balzo in avanti di uno dei settori al contempo più rilevante e delicato per la vita delle persone, quello sanitario. Ciò è evidente non solo in virtù delle numerose possibilità attualmente offerte dalla tecnologia, ma anche per la necessità di poter fare affidamento su uno strumento profondamente strutturato e resiliente, capace di erogare servizi in modo intuitivo, e di fronteggiare efficacemente potenziali situazioni di crisi. In quest’ottica, una delle soluzioni identificate è la Telemedicina, un paradigma che mira alla digitalizzazione dei processi che coinvolgono pazienti e professionisti attraverso l’impiego delle tecnologie informatiche e della comunicazione.

Il progetto protagonista dell’elaborato in questione appartiene a questo contesto, e si configura come un’attività di sviluppo che ha portato all’implementazione di nuovi componenti all’interno dell’interfaccia utente dell’applicazione web “OMNIA”. La relazione rappresenta un’analisi della suddetta attività progettuale e l’esposizione delle diverse sezioni realizza una progressione: nelle prime due vengono affrontate le tematiche teorico/nozionistiche di alto livello; la terza e la quarta si rapportano con aspetti più pragmatici e tecnici di basso livello; infine, nella quinta viene fornita una trattazione di ciò che di pratico è stato svolto.

“OMNIA” rappresenta uno dei moduli di cui si compone l’articolata piattaforma per la digitalizzazione dei flussi operativi aziendali che Link-Up sta sviluppando per Cerba HealthCare, un gruppo francese specializzato in diagnostica ambulatoriale che mira a distinguersi anche in Italia per la qualità e la modernità dei servizi offerti in ambito medico-sanitario. Una più dettagliata descrizione della “cornice” del progetto, ovvero di tutti quegli aspetti che permettono una collocazione tematica dell’applicativo sul quale si è operato, è fornita all’interno del primo capitolo.

Come anticipato, il modulo protagonista dell’elaborato, così il come la piattaforma nella sua interezza, rappresenta un’applicazione web, o meglio, una *Single-Page Application*. Le web app di questo genere si avvalgono di un funzionamento che simula quello di una singola pagina web, e sono molto popolari grazie all’esperienza utente, alla rapidità di sviluppo e alla riduzione dei costi che sono in grado di offrire. Questi e altri concetti sono affrontati nel secondo capitolo, sezione atta alla descrizione delle caratteristiche che il prodotto realizzato per Cerba HealthCare presenta.

L’attività di sviluppo condotta con il team di Link-Up è avvenuta tramite il lavoro da remoto,

---

il quale è stato declinato secondo la metodologia agile in combinazione con il *framework Scrum*. Con l’impiego di questo *modus operandi*, i compiti previsti dalle evolutive programmate per “OMNIA” sono stati organizzati in *sprint*, delle brevi finestre temporali dedicate allo sviluppo di una specifica funzionalità dell’applicativo, pensate per agevolare il lavoro degli sviluppatori e l’interazione con il committente. Le strategie adottate per la gestione della cooperazione con il team sono descritte nel terzo capitolo, sezione che si discosta dalle precedenti per *focus* tematico, e costituisce un autentico spartiacque all’interno della struttura generale dell’elaborato. A partire da qui, il fulcro della trattazione diviene l’attività di sviluppo.

L’interfaccia utente dell’applicazione web “OMNIA” è stata realizzata per lo più con l’utilizzo delle librerie React e Material UI, due strumenti per lo sviluppo lato *front-end* molto popolari nella community grazie a pregi come semplicità, supporto, velocità, flessibilità e modularità. In conseguenza a ciò, esse rappresentano le due soluzioni tecnologiche principali con cui mi sono relazionato in prima persona per la realizzazione delle implementazioni, nonché i due argomenti fondamentali trattati nel quarto capitolo. Questo prosegue nella descrizione, iniziata nella precedente sezione, delle soluzioni impiegate nel corso dell’attività di sviluppo, incentrandosi però sugli aspetti tecnologici.

Le diverse implementazioni realizzate sono state caratterizzate da un differente livello di complessità, ed hanno avuto a che fare, in particolare, con l’introduzione di una serie di sezioni a supporto del team di *marketing*. Le *feature* più importanti sviluppate sono state elencate all’interno del quinto e ultimo capitolo, sezione nella quale viene mostrato l’*output* dell’attività di sviluppo condotta ai fini del progetto. Al suo interno viene anche descritto il “Flusso di sviluppo standard”, ovvero la procedura tipo seguita per realizzare un’implementazione, comprensiva di tutti gli *step* necessari. L’idea di fornire un punto di riferimento stabile e allo stesso tempo un *fil rouge* in grado di tracciare il percorso attraverso i vari passaggi seguiti, spesso solo parzialmente presenti, è stata motivata dal desiderio di generalizzare il più possibile il processo realizzativo prima di addentrarsi nell’analisi approfondita di ogni singolo caso.

# 1. Inquadramento contestuale

Il primo capito ha lo scopo di contestualizzare il progetto oggetto dell’elaborato fornendone un’iniziale visione d’insieme. Per farlo va a trattarne le caratteristiche di più alto livello, la cui descrizione è doverosa prima di addentrarsi in questioni più teoriche o pratiche. A tal fine, il capitolo incomincia offrendo una panoramica generale dell’area tematica a cui il progetto appartiene, la telemedicina, e del suo substrato aziendale, Link-Up e Cerba HealthCare, per poi proseguire con una sua disamina complessiva che tiene conto degli obiettivi perseguiti nel corso dell’attività di sviluppo.

## 1.1 Telemedicina

Nonostante il mondo in cui viviamo sia caratterizzato da una forte presenza tecnologica in quasi tutti i settori aziendali, alcuni di essi stanno attraversando un’autentica fase di transazione proprio al giorno d’oggi, con scenari che variano da paese a paese. Generalmente, questo tipo di settori o sono protagonisti di una radicale rivoluzione, la quale richiede un intervento diffuso con l’intento di ammodernare dal profondo, oppure hanno già beneficiato dell’impatto della tecnologia, ma solo parzialmente, necessitando quindi di proseguire nel processo di innovazione.

Uno dei settori più ambivalenti, in questo senso, è quello medico: infatti, seppur in alcuni frangenti possa vantare uno dei più alti livelli di modernità, in altri questo campo rimane ancorato a metodologie del passato, specialmente in Italia.

Tuttavia, negli ultimi anni, i tempi sembrano maturi per un importante balzo in avanti. Uno dei fattori che sicuramente ha più contribuito a fornire un decisivo impulso evolutivo è l’emergenza sanitaria causata dal COVID-19: infatti, le caratteristiche del virus e la sua enorme diffusione, sono riusciti a mettere in crisi il settore medico mostrandone chiaramente i limiti attuali, e facendo insorgere l’urgenza di nuovi strumenti.

L’accelerazione dovuta alla pandemia ha portato all’ingresso<sup>1</sup> di una nuova soluzione medica nel Servizio Sanitario Nazionale (SSN) italiano, ovvero della Telemedicina. Poiché essa rappresenta ancora un qualcosa di pionieristico e sperimentale, e, di conseguenza, di avvolto da

---

<sup>1</sup>La Telemedicina ha fatto il suo ingresso nel SSN nel dicembre 2020. In altri paesi europei, fra cui Svezia, Norvegia, Regno Unito e Spagna è molto diffusa da tempo.

una fitta nube di confusione, è bene fare chiarezza.

La Telemedicina è definita come:

«[...] una modalità di erogazione di servizi di assistenza sanitaria, tramite il ricorso a tecnologie innovative, in particolare alle *Information and Communication Technologies* (ICT), in situazioni in cui il professionista della salute e il paziente [...] non si trovano nella stessa località. La Telemedicina comporta la trasmissione sicura di informazioni e dati di carattere medico [...] per la prevenzione, la diagnosi, il trattamento e il successivo controllo dei pazienti. I servizi di Telemedicina vanno assimilati a qualunque servizio sanitario diagnostico/terapeutico. Tuttavia la prestazione in Telemedicina non sostituisce la prestazione sanitaria tradizionale nel rapporto personale medico-paziente, ma la integra per potenzialmente migliorare efficacia, efficienza e appropriatezza»[23].

## 1.2 Link-Up e Cerba HealthCare

Link-Up è un'azienda specializzata nello sviluppo di *software serverless*, di tipo *cloud-native*, e nella progettazione di soluzioni di *Data Analytics* e Intelligenza Artificiale che sfruttano i dati ricavati da dispositivi eterogenei connessi all'*Internet of Things* (IoT).

Questa realtà opera prevalentemente nel settore medico-sanitario e offre ai suoi clienti strategie innovative e supporto orientato alla trasformazione digitale.

Uno dei contesti aziendali di maggiore rilievo con cui Link-Up sta collaborando in maniera attiva è Cerba HealthCare, un gruppo internazionale francese leader nell'ambito della diagnostica ambulatoriale che recentemente è stato protagonista di una significativa espansione in Italia con l'acquisizione di numerose strutture mediche.

La stimolante missione che Cerba sta affrontando consiste nell'offrire un'assistenza sanitaria digitalizzata capace di mettere al centro la persona, semplificando l'esperienza di pazienti, aziende e operatori tramite strumenti innovativi, come, ad esempio, cartelle cliniche elettroniche, servizi di telemedicina, prenotazioni online, etc...

## 1.3 OMNIA

Ad oggi, Link-Up coopera con Cerba HealthCare per la realizzazione di un complesso applicativo per la digitalizzazione dei flussi operativi aziendali, che mira a incidere sulla *performance* dei professionisti della sanità ammodernando i processi che li vedono coinvolti e agevolando le gestione delle attività e delle strutture sparse sul territorio italiano.

L'applicativo si configura come una piattaforma composta da moduli che digitalizzano i flussi e la gestione dei diversi verticali in cui è attiva Cerba HealthCare Italia. Ciascun modulo prevede l'interazione di diverse figure professionali, anche in simultanea, e la possibilità di svolgere azioni differenti a seconda del ruolo ricoperto.

A partire da settembre 2022, il team di sviluppo di Link-Up è tornato a focalizzarsi su “OMNIA”, uno dei moduli più importanti della piattaforma di digitalizzazione. Il modulo “OMNIA” permette di gestire in maniera capillare le strutture di Cerba HealthCare Italia sul territorio, nonché digitalizzare i flussi dell'attività logistica a esse legata.

Nello specifico, sono state programmate numerose evolutive che riguardano per lo più l'aggiunta di nuove funzionalità richieste dal committente, in particolare a supporto del team di *marketing*, così da rendere possibile attraverso il modulo “OMNIA” la gestione di contenuti presentati all'interno del sito web aziendale, con l'obiettivo di centralizzare e unificare la gestione delle informazioni utili agli utenti dei centri di Cerba HealthCare Italia.

“OMNIA”, così come gli altri moduli che compongono la piattaforma sviluppata da Link-Up, rappresenta un'articolata *Single-Page Application* la cui interfaccia utente è stata sviluppata tramite le librerie React e Material UI (MUI).



## 2. Applicazioni web

Il secondo capitolo ha lo scopo di fornire una descrizione delle qualità che caratterizzano il progetto oggetto dell’elaborato. Per farlo va a trattare una serie di nozioni teoriche che consentono di giustificare le scelte strutturali a corredo dell’applicativo. A tal fine, il capitolo offre una rapida introduzione sulle applicazioni web, per poi esporre dei dettagli in merito alla loro storia, alle loro peculiarità architettoniche, e alle loro tipologie esistenti.

### 2.1 Principi generali

Con l’espressione “applicazione web” (in inglese *Web application*, abbreviato in *Web app*) si fa riferimento a un *software* archiviato su un server, eseguito all’interno di un browser, che mette a disposizione dell’utente una serie di funzionalità interattive simili a quelle di un’applicazione nativa o desktop.

La natura delle web app è implicitamente multipiattaforma. Diversamente dalle applicazioni native e desktop, quelle web sono caratterizzate dall’assenza di un legame forte con il sistema operativo: infatti, esse non necessitano di essere installate, e le uniche condizioni necessarie e sufficienti affinché queste possano essere utilizzate sono la disponibilità di un browser in grado di supportarle<sup>1</sup> e la connessione a un network, richiesta nella maggioranza dei casi<sup>2</sup>. Quindi, le web app possono funzionare a prescindere dal dispositivo in uso, indipendentemente dal fatto che si tratti di uno smartphone, di un tablet o di un desktop.

Le applicazioni web, a differenza di quelle native e desktop, sono più sicure e riducono la possibilità di incappare in bug. Esse non richiedono aggiornamenti manuali la cui mancata esecuzione potrebbe esporre a potenziali falle; quando si accede a esse si ha automaticamente a disposizione la loro versione più sicura e/o aggiornata.

In aggiunta, essendo svincolate dal sistema operativo in uso, le web app non devono essere programmate più volte in diversi linguaggi e ciò si traduce sia in una maggior rapidità di sviluppo, che nella necessità di meno programmatori e competenze.

Inoltre, le applicazioni web propongono una serie di funzionalità circoscritte a un determinato ambito incastonate in una *User Experience* (UX) più ricca e dinamica, la quale, pur rimanendo

---

<sup>1</sup>Tutti i browser moderni sono in grado di supportare l’esecuzione di applicazioni web.

<sup>2</sup>Seppur le web app necessitino quasi sempre di una connessione internet, alcune di esse possono essere eseguite grazie alla rete intranet aziendale, o addirittura offline.

inferiore se comparata a quella delle applicazioni native<sup>3</sup>, offre una maggior possibilità di interazione rispetto a quella che caratterizza i siti web.

Poiché non si rassegnano all’essere un qualcosa di meramente statico-informativo, le web app sono presto divenute molto popolari, ed è realistico credere che, in un futuro non troppo lontano, esse contribuiranno all’affermazione di un web ancor più dinamico e interattivo.

Alcune fra le applicazioni web più famose ad oggi accessibili sono: *Facebook, Instagram, Twitter, Linkedin, Amazon, e-Bay, Gmail, Outlook, Google Drive, Microsoft Office 365, Slack, Paypal, Google Maps, Netflix, Amazon Prime Video*, etc...

In definitiva, le web app si contraddistinguono per il loro essere *user-friendly* per l’utente, per gli sviluppatori e per l’azienda responsabile della loro realizzazione; a differenza delle applicazioni native e desktop, le web app sono più accessibili, flessibili e sicure, e il loro sviluppo richiede meno tempo e risorse economiche.

## 2.2 Cenni storici

La storia delle applicazioni web coincide con quella del *World Wide Web* (WWW) (anche conosciuto più semplicemente come *Web*) e ha inizio nel 1980, quando l’informatico britannico Tim Berners-Lee, presso il CERN (acronimo del francese *Conseil Européen pour la Recherche Nucléaire*), si mise all’opera per la creazione di uno strumento che permettesse la «[...] condivisione automatizzata delle informazioni tra gli scienziati delle università e degli istituti di tutto il mondo»[6]. Nel 1989 venne pubblicato uno studio che descriveva un «[...] “progetto ipertestuale” chiamato “WorldWideWeb” in cui una “ragnatela” di “documenti ipertestuali” poteva essere visualizzata da “browser”»[6]. L’intuizione rivoluzionaria fu quella di utilizzare un applicativo lato client (il browser web) per reperire contenuti di natura ipertestuale, interconnessi fra loro attraverso link e raggiungibili grazie a internet, immagazzinati in un *software* lato server. Quello stesso anno, inoltre, iniziarono i lavori di sviluppo degli applicativi, i quali vennero affiancati da quelli di definizione di nuovi standard e protocolli per la condivisione di contenuti, ovvero l’*HyperText Markup Language* (HTML) e l’*HyperText Transfer Protocol* (HTTP). Lo sviluppo del *software* del primo server e del primo browser venne completato l’anno successivo, nel

<sup>3</sup>Il livello di interattività offerto dalle web app è inferiore a quello delle applicazioni native poiché le prime, contrariamente delle seconde, non potendo essere cucite su misura all’*hardware*, non riescono a sfruttare a pieno le funzionalità del dispositivo in uso. Inoltre, se paragonata a quella delle applicazioni native, l’esperienza utente delle applicazioni web è inferiore soprattutto in termini di *performance*.

1990, data in cui venne pubblicata anche la primissima pagina web. Seppur essa fosse inizialmente visibile solamente ai membri interni del CERN, da quel giorno, l'espansione del Web non ebbe più fine: infatti, prima, nel 1991, venne messo a disposizione dell'intera comunità scientifica, e poi, a partire dal 1993, del mondo intero.

A pari passo con l'evoluzione tecnologica e informatica, negli ultimi 30 anni il Web è cambiato notevolmente, e le sue declinazioni affermatesi nel corso del tempo sono state caratterizzate da un costante aumento dell'interattività e della connettività offerta:

- **Web 1.0** (1990/93 - 2000 ca.): il primo paradigma, anche detto Web statico, o *read-only Web*, è dotato di una natura prettamente statico-informativa e contenutistica. Le pagine, costituite da puro HTML, e, all'occasione, da qualche sprazzo di *Cascading Style Sheets* (CSS), non offrono alcuna possibilità d'interazione. Il Web 1.0 è unidirezionale: l'utente può solo consultare passivamente i contenuti "pre-inscatolati" dalla ristretta schiera di persone che conoscono gli strumenti per la manipolazione delle pagine web. Tipici di questo stadio del Web sono i siti di informazioni e notizie, i primi e-commerce e le piattaforme per lo scambio di e-mail;
- **Web 2.0** (2000 - 2006 ca.): il secondo paradigma, anche detto Web dinamico, o *read-write Web*, è dotato di una natura decisamente più dinamica e partecipativa rispetto al predecessore. Le pagine, arricchite con il linguaggio di scripting JavaScript, offrono la possibilità d'interagire con esse, ad esempio, condividendo risorse; altre tecnologie risalenti a questo periodo determinanti per l'evoluzione dell'interattività sono *Adobe Flash Player (Flash)* e *Asynchronous JavaScript and XML (AJAX)*. Il Web 2.0 è bidirezionale: l'utente può sia fruire di contenuti, che produrre attivamente i suoi per condividerli con altre persone. Tipici di questo stadio del Web sono i blog, i forum e i social network;
- **Web 3.0** (2006 - oggi): il terzo paradigma, anche detto Web semantico, o *read-write-execute Web*, è dotato di una natura semantica incentrata sui dati. Le pagine, elaborate in modo più raffinato dai motori di ricerca, sono a disposizione dell'utente in maniera più precisa grazie a interrogazioni meno complesse. Le tecnologie impiegate, per la maggior parte evoluzione di quelle inventate in passato (ad esempio, l'HTML5), permettono al programma di "comprendere" le risorse presenti in rete attraverso la trattazione automatica di metadati. Il Web 3.0 è caratterizzato anche dall'uso dell'*Artificial Intelligence* (IA), dei potenti algoritmi in grado di proporre contenuti *ad hoc* all'utente, e da un grande aumento

della cura riposta nella UX delle applicazioni web, prestando attenzione a concetti quali responsività e usabilità, dato l'accresciuto utilizzo di dispositivi *mobile*. Tipici di questo stadio del Web sono le moderne web app citate nella sezione 2.1.

In definitiva, è possibile affermare che le web app siano nate, almeno concettualmente, negli anni del Web 1.0. Nonostante ciò, esse si sono evolute, e quelle di cui facciamo quotidianamente uso ai giorni nostri appartengono a ciò che viene identificato come Web 3.0.

## 2.3 Architettura

Da un punto di vista architettonico, le applicazioni web appartengono alla famiglia dei sistemi distribuiti<sup>4</sup>, dei sistemi informatici la cui più rilevante proprietà è la delocalizzazione del carico computazionale. I sistemi distribuiti sono caratterizzati dalla presenza di molteplici processi interconnessi fra loro, che sono eseguiti in parallelo su nodi di calcolo diversi, e condividono un obiettivo comune.

Parlare dell'architettura delle web app significa parlare dell'architettura *three-tier*, una filosofia di progettazione classica nell'ambito dell'ingegneria del *software*, e prendere in considerazione anche le architetture che hanno portato alla definizione di diverse tipologie applicazioni web.

### 2.3.1 Livelli

Nonostante le web app possano essere molto differenti fra loro in termini di architettura *software* impiegata, la maggior parte di esse sono riconducibili a un'architettura a tre livelli, o strati (in inglese *three-tier*). Essa rappresenta un particolare tipo di architettura a più livelli (in inglese *multi-tier* o *n-tier*) considerata l'evoluzione della tradizionale architettura a due livelli (in inglese *two-tier*) di tipo client-server.

L'architettura *three-tier* si avvale di una segmentazione dell'applicativo in moduli che vanno a rispecchiare le sue funzioni. Ognuno dei tre livelli comunica con quello più prossimo riproducendo il paradigma client-server, e quelli di cui si compone sono:

- **Presentation tier:** il primo livello, il più alto, è responsabile del *rendering*, letteralmente della “presentazione”, a schermo dell'interfaccia utente dell'applicativo, in inglese *User Interface*.

<sup>4</sup>I sistemi distribuiti rappresentano l'evoluzione dei sistemi centralizzati poiché sono caratterizzati da resilienza, scalabilità, eterogeneità, disponibilità e condivisione delle risorse.

*Interface* (UI). Ciò avviene grazie al browser web, il *software* che permette la fruizione di contenuti e servizi da parte dell’utente attraverso l’invio di richieste, e l’interpretazione dei risultati restituiti dal server web. In definitiva, il livello di presentazione si rapporta con dinamiche lato client (in inglese *client-side*), anche identificate come *front-end* poiché immediatamente visibili. Alcuni dei più importanti linguaggi utilizzati per la creazione di interfacce utente sono, ad esempio, HTML, CSS, JavaScript, e più recenti *framework* basati su JavaScript, come, ad esempio, *React*, *Angular* e *Vue*;

- **Business tier:** il secondo livello, l’intermedio, è responsabile dell’effettivo funzionamento dell’applicativo, e dunque della logica applicativa<sup>5</sup> (anche detta “logica di *business*”) deputata all’elaborazione delle richieste effettuate dall’utente lato client. Essa risiede in un *middleware*, ed è identificata come *back-end* poiché non immediatamente visibile all’utente. Alcuni dei più importanti linguaggi impiegati per la gestione della logica di funzionamento dell’applicativo sono, ad esempio, Python, PHP e Java;
- **Data tier:** il terzo livello, il più basso, è responsabile dell’immagazzinamento, della manipolazione e dell’interrogazione dei dati presenti sul *DataBase* (DB) grazie alla gestione delle richieste provenienti dalla logica applicativa. Il livello dei dati può risiedere anch’esso sul *middleware*, oppure sul server.

Un sistema che implementa una simile strutturazione permette agli sviluppatori di maneggiare il *software* accedendo solamente a sezioni specifiche dell’applicativo.

### 2.3.2 Tipologie

Le applicazioni web presenti sul mercato possono essere suddivise in:

- **Multiple-Page Application (MPA):** il loro funzionamento è basato su molteplici pagine, ognuna delle quali viene reperita *ex novo* in seguito a una richiesta effettuata al server ogni volta che l’utente effettua un click su un link. Una MPA è formata da pagine HTML, la cui interattività dipende dall’esecuzione di codice JavaScript. Poiché, prima di essere restituita al browser e visualizzata, ogni pagina deve essere caricata e renderizzata all’interno del server, si parla di *Server-Side Rendering* (SSR).

Le MPA rappresentano il paradigma di applicazione web più tradizionale; esse altro non

---

<sup>5</sup>Il *Business tier* è il livello assente nell’architettura *two-tier*.

sono che i siti web classici, come i siti di e-commerce, i portali di notizie e i siti di informazioni;

- **Single-Page Application (SPA)**: il loro funzionamento avviene in una singola pagina, la quale viene reperita nella sua interezza attraverso un'unica richiesta HTTP iniziale e aggiornata dinamicamente attraverso le molteplici richieste effettuate al server all'interagire dell'utente con determinate porzioni della web app. Al posto di richiedere una nuova pagina ogni volta che è necessario visualizzare nuovi dati, le SPA mantengono tutte le porzioni della web app non destinate a cambiare (che sono state ottenute grazie alla prima chiamata, la principale) ed effettuano richieste specifiche per popolare solo le aree che devono essere aggiornate.

Le SPA sono molto apprezzate sia da coloro che le utilizzano, che da coloro che le sviluppano. Da un lato l'esperienza utente offerta è più fluida, reattiva e interattiva: restituendo solo i contenuti richiesti di volta in volta, evitando di ricaricare pagine pregne di contenuti superflui e facendo ampio uso di codice JavaScript per la gestione di una complessa interattività, le SPA offrono delle prestazioni e una dinamicità nettamente superiori a quelle delle MPA, avvicinandosi alle applicazioni native e desktop. Dall'altro, il lavoro dello sviluppatore è più rapido: utilizzando in più situazioni diverse gli stessi componenti, in una SPA il codice può essere riutilizzato, sono meno gli elementi da testare, la responsività è più facile da gestire, e lo scambio di dati/informazioni può essere monitorato con meno sforzo.

I difetti principali di questo tipo di applicazione web sono il tempo di caricamento iniziale, potenzialmente lungo a causa dell'ingente mole di dati che devono essere reperiti dalla prima richiesta, e la difficile gestione della *Search Engine Optimization* (SEO).

Questo tipo di applicazioni web vengono sviluppate grazie a *framework* JavaScript come *React*, *Angular* e *Vue*;

- **Progressive Web Application (PWA)**: il loro funzionamento imita quello delle applicazioni native e desktop. Infatti, pur rimanendo eseguibili anche solamente sul browser, le PWA possono essere installate rapidamente tramite un pulsante posto nel browser, senza il bisogno di uno *store*.

Grazie a una migliore integrazione con l'*hardware* e il *software* del dispositivo in uso, questo tipo di applicazioni web è dotato di un ottimo livello di interattività. Fra le funzioni più interessanti delle PWA vi sono l'accesso a periferiche come il GPS e la fotocamera, la

possibilità di utilizzare funzioni native come la scansione di codici QR, le notifiche *push*, e la possibilità di utilizzo offline;

- ***Isomorphic Web Application (IWA)***: la loro esecuzione può avvenire sia lato client che lato server. In altre parole, le stesse funzionalità e la stessa logica possono essere eseguite da entrambi i lati a seconda delle esigenze.

Poiché le richieste possono essere gestite sia sul server che sul client, l'innata duttilità delle IWA si traduce in una minore latenza e in una maggiore scalabilità. Ciò permette di offrire un'esperienza utente più fluida e coerente, poiché la logica e le funzionalità sono condivise tra il server e il client.

Se le MPA e le SPA rappresentano dei paradigmi piuttosto affermati nell'ambito dello sviluppo web, le PWA e le IWA costituiscono delle soluzioni che hanno ancora un grande margine di crescita e diffusione.



# 3. Soluzioni organizzative

L'obiettivo che il terzo capitolo si pone è quello di spiegare come il coordinamento con il team di sviluppo sia avvenuto. Per farlo va ad analizzare le soluzioni adottate a livello organizzativo, ovvero tutti quegli strumenti che hanno permesso la pianificazione delle operazioni necessarie alla realizzazione del progetto nel corso dei mesi. A tal fine, il capitolo parte fornendo una panoramica generale del concetto di *smart working* e di metodologia agile, per poi proseguire con un'analisi più dettagliata del *framework Scrum*, un particolare approccio alla metodologia agile stessa.

## 3.1 *Smart working*

Il progetto è stato realizzato quasi interamente attraverso il lavoro agile, da remoto, al quale si è soliti riferirsi grazie alla più nota espressione, ormai entrata a far parte del linguaggio comune, “*smart working*”.

Questo è definito come:

una «[...] modalità di esecuzione del rapporto di lavoro subordinato [...] con forme di organizzazione per fasi, cicli e obiettivi e senza precisi vincoli di orario o di luogo di lavoro, con il possibile utilizzo di strumenti tecnologici per lo svolgimento dell'attività lavorativa. La prestazione lavorativa viene eseguita, in parte all'interno di locali aziendali e in parte all'esterno senza una postazione fissa, entro i soli limiti di durata massima dell'orario di lavoro giornaliero e settimanale [...]»[10].

La forma di telelavoro in questione è stata adottata per la prima volta in modo stabile a causa dell'emergenza sanitaria dovuta al COVID-19, nel 2020. Infatti, poiché la pandemia ha costretto più volte a severi *lockdown*, è stato necessario trovare una contromisura che permettesse di continuare a svolgere tutte le operazioni di *routine* lavorativa da remoto.

In virtù della straordinaria bontà di alcuni suoi aspetti positivi, come, ad esempio, «[...] la possibilità di organizzare e programmare meglio il lavoro; [...] la maggiore disponibilità di tempo per sé e per la propria famiglia; [...] il lavorare in un clima di maggiore fiducia e responsabilizzazione [...]» e «[...] un modo di lavorare più stimolante»[7], lo *smart working* è poi permasto anche una volta terminato il periodo critico della pandemia. Ad ora, costituisce una soluzio-

ne importante nella vita di molte persone, e uno standard per numerose realtà che operano nel settore *Information Technologies* (IT)

Seppur il lavoro da remoto abbia rappresentato la soluzione adottata preponderantemente per la pianificazione delle attività e la coordinazione con il team di sviluppo con il quale è stato realizzato il progetto, va menzionata la presenza di un unico appuntamento settimanale in presenza, quello del martedì.

## 3.2 Metodologia agile

L'attività (svolta in *smart working*) condotta assieme al team di sviluppo di Link-Up è stata declinata secondo la metodologia agile.

Questa può essere definita come:

«[...] un approccio allo sviluppo del *software* basato sulla distribuzione continua di *software* efficienti creati in modo rapido e iterativo»[27].

La metodologia agile, anche nota come sviluppo del *software* agile (in inglese *Agile Software Development*, ASD), consiste in un insieme di metodi caratteristici dell'ambito sviluppo *software* apparsi nei primi anni 2000 ispirati ai principi contenuti nel “Manifesto per lo sviluppo agile del *software*” [5].

Tale approccio, conosciuto inoltre come *Lightweight*, nasce in contrapposizione alla metodologia di sviluppo a cascata (in inglese *Waterfall*), una delle più solide e antiche, e in generale a tutte le procedure di sviluppo *software* tradizionali, dette *Heavyweight*.

Al contrario degli approcci di tipo classico, strutturati in modo sequenziale, dove ogni fase deve essere completata prima di poter passare alla seguente, e in cui ci si interfaccia con il committente sporadicamente<sup>1</sup>, l'approccio agile è caratterizzato da una notevole flessibilità.

All'interno del “Manifesto per lo sviluppo agile del *software*”, durante la definizione dei principi alla base della nuova metodologia, gli sviluppatori si sono discostati dalla linearità e dalla rigidità tipica degli approcci del passato affermando la priorità di quattro concetti fondamentali<sup>2</sup>:

<sup>1</sup>Gli approcci di tipo classico prevedono che l'interazione con il committente avvenga solamente a prodotto realizzato. Ciò è potenzialmente molto pericoloso: può condurre al fallimento o a grandi sprechi di tempo e di denaro.

<sup>2</sup>Tutti gli altri principi propri delle metodologie tradizionali non vengono dimenticati o estromessi ma solamente considerati di minore importanza.

- «Gli individui e le interazioni più che i processi e gli strumenti;
- Il software funzionante più che la documentazione esaustiva;
- La collaborazione col cliente più che la negoziazione dei contratti;
- Rispondere al cambiamento più che seguire un piano»[5].

La maggior attenzione riservata agli elementi sopra elencati ha portato a una segmentazione del progetto in parti più piccole, caratterizzate da lassi temporali di durata inferiore.

Il paradigma agile, infatti, ammette la divisione del progetto in micro fasi, chiamate *sprint*, ciascuna delle quali è dedicata all'implementazione di una porzione di *software* ridotta, come, ad esempio, una specifica funzionalità (in inglese *feature*).

Una volta terminato lo sviluppo delle implementazioni programmate per uno *sprint*, quanto prodotto viene somministrato immediatamente al cliente. L'obiettivo è quello di ricevere un *feedback* che permetta di valutare in tempi brevi il livello di soddisfazione del committente, per raggiungere un miglioramento continuo nel corso di ogni fase di sviluppo.

Ciò costituisce a tutti gli effetti un'evoluzione delle metodologie di sviluppo tradizionali: a differenza di quanto accadeva in passato, l'innovativo paradigma permette ora la conduzione di più sequenze (di sviluppo e di test) continue e simultanee nel corso di un maggior numero di iterazioni che avvengono in lassi di tempo più brevi.

Inoltre, è importante sottolineare come il superamento e il miglioramento dei sistemi classici non si limiti alla “sfera” contenutistica, riguardando anche quella concettuale. Nel suo abbandono della rigidità, infatti, la metodologia agile non si configura più come un insieme di regole da seguire in modo ferreo, bensì come un «[...] approccio alla collaborazione e ai flussi di lavoro fondato su una serie di valori in grado di guidare il [...] modo di procedere»[27].

Conseguenza di ciò è un nuovo sguardo alle relazioni lavorative. Operare rispettando delle “linee guida” (e non più delle regole) significa godere di una maggiore autonomia organizzativa e di una comunicazione con il cliente meno mediata, frequente nel corso di tutto il processo di sviluppo dell'applicativo attraverso dei referenti aziendali.

### **3.3 Framework Scrum**

Procedere adottando la metodologia agile non significa svolgere attività nello stesso modo in tutte le situazioni. Varie sono infatti le interpretazioni esistenti di quest'approccio allo sviluppo

del *software*. In gergo, (più che di “interpretazioni”) si parla di *framework*, e fra i più conosciuti e utilizzati troviamo: *Scrum*, *Kanban* ed *Extreme Programming*.

*Scrum*, in particolare, rappresenta il *framework* impiegato per la gestione del progetto protagonista dell’elaborato, e, più in generale, il tipo di metodologia agile adottata nella stragrande maggioranza dei progetti di sviluppo *software* complessi.

Questo viene definito come:

«[...] un *framework* che aiuta i team a lavorare insieme. Proprio come una squadra di rugby (a cui deve il nome) che si allena per un grande evento sportivo, il *framework* *Scrum* incoraggia i team a imparare attraverso l’esperienza, a organizzarsi in modo autonomo mentre lavorano su un problema e a riflettere sui risultati conseguiti e sugli insuccessi per migliorare continuamente»[1].

Il termine “*Scrum*”, in inglese, ha a che fare con il rugby ed è traducibile in “mischia”. La filosofia di questo *framework* ruota interamente attorno a questo sport, ispirandosi fortemente sia alla collaborazione che deve legare tutti i membri di una squadra, sia al modo in cui viene affrontata una situazione di gioco, in mischia per l’appunto.

Contrariamente a quanto potrebbe sembrare, infatti, il lavoro dello sviluppatore non costituisce un qualcosa di individuale, bensì un’attività che prevede una forte interazione, in cui si opera in gruppo per la realizzazione di un prodotto. La grande mole di lavoro con la quale ci si deve misurare viene divisa in *sprint*, brevi cicli temporali dedicati allo sviluppo di una singola parte dell’applicativo, all’interno dei quali l’implementazione da realizzare viene scomposta in una serie di piccoli compiti (in inglese *task*) affrontati in blocco. Al termine di ogni *sprint* il prodotto viene rilasciato e mostrato al cliente.

Seppur la durata canonica degli *sprint* vada dalle due alle quattro settimane, l’attività di sviluppo condotta assieme al team di Link-Up ha previsto un rilascio settimanale fissato per il giovedì.

### 3.3.1 Protagonisti

La pianificazione di un progetto secondo il *framework Scrum* prevede il contributo di tre ruoli ben definiti:

- **Team di sviluppo:** il gruppo di sviluppatori, solitamente dalle dimensioni comprese fra i cinque e i nove membri, che si occupa prima dell’organizzazione (o meglio, dell’auto-

organizzazione) delle attività più importanti richieste dal cliente in *task*, i quali definiscono la mole di lavoro da svolgere durante lo *sprint*, e poi della realizzazione e del *testing* del prodotto.

Nonostante allo stesso progetto possano essere dedicati gli sforzi di più gruppi di sviluppatori, ogni piccolo team è caratterizzato da un'anima interfunzionale. Al loro interno, membri dalle diverse capacità si occupano di attività riguardanti la loro area di competenza evitando, o per lo meno riducendo, la presenza di molteplici team specialistici<sup>3</sup>;

- **Scrum master:** colui che si occupa della corretta comprensione e realizzazione del progetto da parte del team di sviluppatori, e che ha il compito di agevolarne l'attività ove possibile: ovvero il responsabile del metodo;
- **Product owner:** colui che conosce tutti i requisiti specifici del progetto e che gestisce sia la comunicazione con i clienti per la definizione di un prodotto realizzabile, sia quella con il team di sviluppo per l'effettiva realizzazione. Il *product owner* deve guidare il team di sviluppo traducendo quanto richiesto dal committente: ovvero il responsabile del progetto.

Il team di sviluppo di Link-Up con cui è stata svolta l'attività necessaria alla realizzazione del progetto era composto da sei membri. Al suo interno, questi erano raggruppati in due “micro-team” a seconda delle competenze: da un lato vi erano gli sviluppatori *front-end* e dall’altro invece quelli *back-end*. Ciascuna di queste unità era dotata di un leader, una persona responsabile degli sviluppi complessivi legati a quell’area; il leader degli sviluppi *back-end* ricopriva al contempo anche il ruolo di *scrum master*.

### 3.3.2 Eventi

La gestione di un progetto secondo la metodologia agile *Scrum* prevede inoltre degli appuntamenti atti alla continua pianificazione delle attività da svolgere.

I principali sono:

- **Daily scrum:** meeting quotidiano della durata massima di 30 minuti nel quale vengono discusse sia l’attività di sviluppo svolta il giorno precedente, che la pianificazione della giornata stessa. In entrambi i casi, qualora presenti, vengono esposte le criticità che sono

---

<sup>3</sup>Evitare la presenza di molteplici team specialistici significa evitare che il lavoro passi, in modo lento e macchinoso, per ognuno di questi fino al suo completamento.

effettivamente state riscontrate, o quelle che si pensa possano affliggere il flusso delle operazioni.

L’obiettivo principale del *daily scrum* è l’ottimale coordinamento del proprio team di sviluppo grazie a un aggiornamento continuo riguardante il procedere dell’attività di tutti i membri;

- **Sprint planning:** meeting all’interno del quale il *product owner*, *scrum master* e team di sviluppo definiscono la porzione di applicativo protagonista del prossimo *sprint*, e la sua scomposizione in piccoli compiti;
- **Sprint review:** revisione svolta al termine di ogni *sprint* finalizzata a determinare se il lavoro svolto è soddisfacente, e quanto lo è. Questo momento prevede sia la partecipazione del cliente che quella del team di sviluppo;
- **Sprint retrospective:** momento di confronto fra i membri del team per decidere quale sarà il percorso da seguire nel corso degli sviluppi futuri. Sulla base del *feedback* del cliente, viene deciso cosa tenere e cosa rimuovere per migliorare il *modus operandi* e di conseguenza il prodotto dello *sprint* successivo.

L’attività svolta con il team di sviluppo di Link-Up è stata caratterizzata da un *daily-scrum* che ha avuto luogo ogni mattina (a eccezione del martedì<sup>4</sup>) attraverso una video chiamata fissata per le 9:00. Lo *sprint-planning* è avvenuto settimanalmente, con stesse modalità e orario del *daily-scrum*, il venerdì. Infine, *sprint review* e *retrospective*, sono stati svolti, ma non secondo le modalità “standard” discusse poco fa. Infatti, lo *sprint review* ha previsto un confronto solamente fra *scrum master*, *product owner* e cliente; il *feedback* inerente all’attività relativa allo *sprint* analizzato veniva poi comunicato al team di sviluppo dallo *scrum master* nel corso dei meeting mattutini. Lo *sprint retrospective*, invece, non ha avuto un vero e proprio momento dedicato poiché è stato integrato all’interno di altri eventi, ad esempio, durante i *daily-scrum* o lo *sprint planning*.

### 3.3.3 Strumenti

Un’attività di sviluppo, di qualunque natura essa sia, richiede degli strumenti per la pianificazione del flusso di lavoro in grado di coordinare al meglio le attività di tutti i ruoli coinvolti.

---

<sup>4</sup>Come anticipato nella sezione 3.1, il martedì è il giorno in cui le attività sono state compiute in presenza.

Jira costituisce l'applicativo utilizzato per la gestione del progetto oggetto dell'elaborato e, assieme a *Slack*, *Trello*, *YouTrack* e tanti altri, una delle soluzioni più utilizzate da *product owner*, *scrum master* e dagli sviluppatori per l'organizzazione del lavoro in team secondo la metodologia agile *Scrum*.

Questa piattaforma *software* struttura le attività avvalendosi dei seguenti artefatti:

- **Product Backlog:** elenco che racchiude tutti i requisiti specificati dal cliente. Poiché il progetto è in continua evoluzione, lo è anche il *product backlog*; il suo aggiornamento termina solamente quando l'attività di sviluppo è giunta al culmine.  
Il *product owner* gestisce quest'elenco amministrandone il contenuto, segmentando i vari requisiti in piccoli *task*, e curandone l'ordinamento, basandosi sulla priorità delle implementazioni da introdurre;
- **Sprint Backlog:** elenco che racchiude i *task* da svolgere nel corso di uno *sprint*. Si tratta di una stima fatta dal team di sviluppo osservando sia le priorità (specificate nel *product backlog*) che la mole di lavoro richiesto per il completamento dello *sprint*;
- **Incremento:** elenco di tutti i *task* (del *product backlog*) portati a termine, appartenenti sia allo *sprint* corrente che ai precedenti.



# 4. Soluzioni tecnologiche

Il quarto capitolo prosegue in quella che è l'analisi del metodo impiegato con il team di sviluppo, focalizzandosi però su aspetti ancora più pragmatici. Infatti, va a descrivere le soluzioni adottate a livello tecnologico, ovvero tutti quegli strumenti che sono stati concretamente utilizzati durante l'effettiva realizzazione delle implementazioni. A tal fine, il capitolo parte fornendo una sostanziosa descrizione di React e Material UI, le due principali librerie attorno alle quali ruota l'intero progetto, per poi concludersi con una rapida panoramica delle altre incorporate dall'applicativo e utilizzate.

## 4.1 React

React (anche conosciuto come React.js o ReactJS) è una libreria Javascript *open-source* per la creazione di interfacce utente (in inglese *User Interface*, UI) interattive.

Creata e mantenuta da Meta (e già in precedenza da Facebook) e da varie aziende e sviluppatori a partire dal 2013, nel corso degli anni, la libreria ha trovato largo consenso presso la comunità di sviluppatori fino a diventare la più popolare in assoluto nei progetti di sviluppo *software* riguardanti applicazioni web.

Con statistiche che affermano un suo impiego in circa il 60% dei casi, infatti, React surclassa *Angular*, *Vue*<sup>1</sup> e tante altre tecnologie per la creazione di UI in termini di utilizzo e diffusione. React è adottato sia da startup che da grandi compagnie affermate. Fra i più famosi *player* di fama internazionale che hanno impiegato questa libreria possiamo citare, ad esempio, *Facebook*, *Instagram*, *Whatsapp*, *Netflix*, *Spotify*, *Paypal*, *Yahoo*, *Uber*, *AirBnB*, *Groupon* e *Dropbox*.

Le caratteristiche che hanno permesso a React di affermarsi sul mercato e di guadagnarsi un gran numero di sostenitori sono:

- **Semplicità:** React è facile da imparare<sup>2</sup> e aiuta a scrivere codice semplice. Nonostante ciò, la libreria fornisce agli sviluppatori degli strumenti che permettono di gestire l'applicazione nel profondo. I più importanti sono quelli che consentono la gestione dello stato, dell'azione e degli eventi;

---

<sup>1</sup>*Angular* (o *Angular 2+*, evoluzione di *AngularJS*) è stato creato da Google nel 2010. *Vue* (o *Vue.js*), invece, è nato 2016 a opera di Evan You. Entrambi sono dei *framework*.

<sup>2</sup>Possedere una buona conoscenza di JavaScript in partenza permette un più agevole approccio.

- **Supporto:** poiché creato e mantenuto da *Facebook*, adottato da molti marchi noti e dotato di una grande community, React dispone di una vasta documentazione e di un ampio numero di risorse prodotte gratuitamente dagli utenti e reperibili online.

Tutto ciò contribuisce alla semplicità di apprendimento della libreria da un lato, e alla stabilità del *trend* di sviluppo che la vede coinvolta dall'altro;

- **Modularità:** React permette di comporre UI complesse assemblando componenti. Ognuno di questi ha una struttura e un comportamento specifici; una volta creati, i componenti possono essere utilizzati ogni qualvolta è necessario, in qualsivoglia parte dell'applicazione;
- **Flessibilità:** in virtù della sua modularità, React permette la creazione di applicazioni semplici da scalare e mantenere.

React è facilmente integrabile in un progetto già esistente, le sue funzionalità possono essere estese attraverso librerie dedicate e può essere usato sia lato client che server.

Inoltre, è possibile realizzare applicazioni *mobile* multipiattaforma<sup>3</sup> (in inglese *cross-platform*) utilizzando React Native;

- **Performance:** il tempo di *rendering* delle pagine delle applicazioni create usando React è minimo. Attraverso il *Document Object Model* (DOM) virtuale, infatti, vengono renderizzate solamente le porzioni dell'interfaccia che hanno subito dei cambiamenti. Oltre tutto, poiché la libreria ben si presta alla SEO, le applicazioni web create attraverso React sono in grado di ottenere un buon posizionamento all'interno dei motori di ricerca e sono facili da trovare.

React è una libreria dichiarativa<sup>4</sup>. Ciò significa che le UI costruite non vengono modellate agendo direttamente sul DOM e modificando le operazioni necessarie alla costruzione dei componenti, ma manipolando lo stato di questi ultimi.

In definitiva, l'obiettivo che la libreria si pone è quello di costruire interfacce utente modulari, frutto dell'aggregazione di componenti riutilizzabili che ne semplificano la struttura generale.

<sup>3</sup>La applicazioni *mobile* multipiattaforma possono essere eseguite su dispositivi aventi sistemi operativi differenti, quindi, sia su iOS che su Android.

<sup>4</sup>L'approccio opposto è quello imperativo. Il suo *modus operandi* prevede che il browser web venga istruito per la modifica del DOM. *JQuery* è una libreria imperativa.

### 4.1.1 JSX

JavaScript<sup>5</sup>, il linguaggio di programmazione del web per eccellenza, costituisce il nucleo attorno al quale React orbita dal punto di vista tecnologico. I componenti che costituiscono le interfacce utente, infatti, sono scritti grazie a un'estensione della sua sintassi chiamata JSX (acronimo dell'inglese *JavaScript Syntax Extension*), realizzata appositamente da *Facebook*.

Il JSX non rappresenta *vanilla* JavaScript; poiché i browser web possono leggere solamente codice JavaScript puro, non sono in grado di interpretarlo. Quindi, se un file contiene codice JSX, per essere eseguito deve essere compilato: prima che il file raggiunga il browser web, un compilatore<sup>6</sup> JSX deve “tradurre” tutto il codice JSX in esso contenuto in chiamate a funzioni JavaScript. Come illustrato nel Listing 4.1, questo processo avviene grazie all’invocazione della funzione `createElement()`.

```
1 ReactDOM.createElement(component, props, ...children);
```

Listing 4.1: Struttura della funzione `createElement()`

Di seguito, grazie al Listing 4.2 e al Listing 4.3 viene illustrato ciò che accade durante la trasposizione da JSX a JavaScript puro.

```
1 function Hello() {
2   return <h1>Hello world</h1>;
3 }
```

Listing 4.2: Componente scritto in JSX

```
1 function Hello() {
2   return React.createElement('h1', null, 'Hello world');
3 }
```

Listing 4.3: Componente scritto in JavaScript puro

Data la sua natura di estensione sintattica, «[...] JSX produce elementi React [...]»[13]: il suo scopo è la creazione di elementi del DOM che vengono poi renderizzati all’interno del

---

<sup>5</sup>Una delle caratteristiche distintive di JavaScript è la sua grande estensibilità e, di conseguenza, adattabilità. Infatti, come dimostrato dallo stesso React, grazie all’impiego di librerie e *framework* appositamente sviluppati, JavaScript è in grado di offrire un’ampia gamma di soluzioni, di diverso tipo, ai problemi da risolvere.

<sup>6</sup>Il compilatore dei progetti basati su JavaScript è *Babel* e la sua impostazione avviene in modo automatico alla creazione di un nuovo progetto React.

DOM di React.

Poiché la funzione `render()` può restituire solamente un nodo, una fra le più importanti regole sintattiche del JSX prevede che, affinché il codice venga compilato senza errori, un'espressione debba essere contenuta all'interno di un solo elemento<sup>7</sup>.

Il codice JSX ricorda molto quello HTML. Le unità più semplici sono chiamate “elementi”; come illustrato dal Listing 4.4, gli elementi possono essere inseriti in ogni posto in cui possono essere inserite le espressioni JavaScript (ad esempio, in variabili, funzioni, `array`, oggetti...).

```
1 const element = <h1>Hello world</h1>;
```

Listing 4.4: Variabile contenente un elemento JSX

Gli elementi JSX possono essere annidati all'interno di altri elementi JSX<sup>8</sup> e avere degli attributi. La loro sintassi prevede che si utilizzi un nome, seguito da un segno di uguale (=), seguito da un valore. Il valore deve essere racchiuso fra virgolette.

Nonostante la grande similitudine a livello grammaticale, esistono delle differenze fra JSX e HTML. Infatti, «dal momento che JSX è più vicino al JavaScript che all'HTML, React DOM utilizza la notazione camelCase nell'assegnare il nome agli attributi, invece che quella utilizzata normalmente nell'HTML, e modifica il nome di alcuni attributi. Ad esempio, `class` diventa `className` in JSX e `tabIndex` diventa `tabIndex`»[13]. La parola `class`, nello specifico, non può essere utilizzata poiché rappresenta una parola riservata in JavaScript. Il problema viene risolto quando il codice JSX viene renderizzato, con la trasformazione degli attributi `className` in `class`.

Un altro punto di discontinuità riguarda i tag autoconclusivi (in inglese *self-closing tags*<sup>9</sup>). A differenza dell'HTML, in cui questo tipo di tag è costituito dal nome del tag racchiuso all'interno delle parentesi angolari (ad esempio, `<br>`), nel JSX la sintassi è la medesima ma con l'aggiunta dello slash<sup>10</sup> (/) prima della parentesi angolare finale.

Il codice inserito fra i tag di un elemento JSX viene considerato come JSX, non come normale JavaScript. Per fare in modo che ciò non accada lo si deve racchiudere fra parentesi graffe. Poiché il codice JavaScript iniettato nel JSX fa parte dello stesso ambiente del resto del Java-

<sup>7</sup>In altre parole, il tag iniziale e quello finale devono appartenere allo stesso elemento.

<sup>8</sup>Quando degli elementi JSX sono annidati all'interno di altri elementi JSX, tutta l'espressione deve essere racchiusa fra parentesi tonde.

<sup>9</sup>I tag autoconclusivi sono quei tag che svolgono sia l'apertura che la chiusura dell'elemento non utilizzando due tag appositi.

<sup>10</sup>L'inclusione dello slash prima della parentesi angolare finale in HTML è opzionale.

Script nel file, è possibile accedere alle variabili all'interno di un'espressione JSX, anche se tali variabili sono state dichiarate all'esterno. Di nuovo, ciò vien fatto richiamando la variabile fra parentesi graffe<sup>11</sup>.

Gli elementi JSX possono avere degli ascoltatori di eventi (in inglese *event listeners*), degli attributi speciali di cui un elemento JSX può essere dotato. Il loro nome è composto dalla parola “on”, seguita dal tipo di evento per il quale si sta ascoltando (ad esempio, `onClick`<sup>12</sup>), e il loro valore deve essere una funzione. Gli attributi ascoltatori di eventi sono numerosi (un elenco che li raggruppa per categoria è riportato al seguente link [17]) e ciascuno corrisponde a un evento specifico.

Dato che all'interno di un'espressione JSX non è possibile iniettare un'istruzione `if`, quando è necessario implementare tale costrutto si può ricorrere a tre alternative. La prima prevede semplicemente che la condizione venga spostata al di fuori del codice JSX<sup>13</sup>. La seconda consiste nell'utilizzo dell'operatore ternario (in inglese *ternary* o *conditional operator*), l'unico operatore JavaScript che ammette tre operandi: una condizione seguita da un punto interrogativo (?), un'espressione da eseguire se la condizione è veritiera seguita da due punti (: ) e, infine, un'espressione da eseguire se la condizione è falsa.

La terza alternativa, invece, prevede che venga impiegato l'operatore logico AND<sup>14</sup> (`&&`); la sua situazione d'utilizzo ideale è quella in cui deve essere eseguita un'azione in un caso, ma nulla nell'altro.

Inoltre, all'interno di React e di JSX vi è un ampio utilizzo dell'iteratore `map()`. Il suo compito è la creazione di liste in modo dinamico per tutti gli elementi contenuti da un array. L'unica regola da rispettare prevede che venga assegnata una *prop key* al componente padre che verrà restituito per tutti gli elementi su cui sta venendo eseguito il *mapping*. Il valore della *prop key* deve essere univoco per ogni elemento generato.

Tutti i concetti appena descritti sono illustrati all'interno del Listing 4.5.

```
1 const expression = (
2   <div>
```

<sup>11</sup>È pratica molto diffusa fra gli sviluppatori utilizzare variabili per l'impostazione degli attributi in JSX.

<sup>12</sup>Anche in questo caso, il JSX, a differenza dell'HTML, adotta la notazione camelCase. In HTML avremmo trovato, ad esempio, `onclick`.

<sup>13</sup>Il codice precedentemente all'esterno della condizione `if` viene messo all'interno della condizione e del corpo dell'istruzione ora spostata all'esterno.

<sup>14</sup>L'operatore ternario e l'operatore logico AND non sono specifici di React ma vengono utilizzati molto spesso dagli sviluppatori nella scrittura del codice.

```

3   <h1 id="title">Title</h1>
4
5   {articles.map(article => (
6     <div key={article.uuid}>
7       <h2 className="article">{article.title}</h2>
8       <br />
9       {article.preview}
10      <br />
11      <div>
12        {article.author && article.author.name}
13        {article.date ? <p>{article.date}</p> : '---'}
14      </div>
15      <button onClick={buttonFunction}>Button</button>
16    </div>
17  ))}
18 </div>
19 );

```

Listing 4.5: Variabile contenente un'espressione JSX

Infine, come riportato dalla documentazione ufficiale, vale la pena precisare che:

«React non obbliga ad utilizzare JSX, ma la maggior parte delle persone lo trovano utile come aiuto visuale quando lavorano con la UI all'interno del codice JavaScript. Inoltre, JSX consente a React di mostrare messaggi di errore e di avvertimento più efficaci»[13].

## 4.1.2 Componenti

L'intero *pattern* architetturale di React ruota attorno al concetto di modularità; tale approccio alla costruzione di interfacce utente è reso possibile attraverso l'impiego di componenti.

Un componente può essere visto come un atomo, come una particella, come una microstruttura dotata di caratteristiche peculiari che è in grado, all'occorrenza, di relazionarsi con l'esterno. Solitamente, il loro compito consiste nel *rendering* di una porzione di HTML. L'affiancamento e l'interazione di più componenti permette la creazione, o meglio, la composizione, di organismi, di macrostrutture, di UI potenzialmente molto complesse sia nella forma che nel comportamento.

React consente agli sviluppatori di creare componenti personalizzati, blocchi di codice indipen-

denti l'uno dall'altro, che possono essere aggregati fra loro e riutilizzati più volte all'interno dell'applicazione.

Una simile filosofia di progettazione rispetta il principio *Don't Repeat Yourself* (DRY), uno dei più importanti quando si parla di programmazione. Banalmente, poiché i componenti vengono scritti una sola volta, viene evitata la ridondanza del codice, il quale ne guadagna in termini di qualità.

È buona prassi che i componenti siano piccoli, rispondendo a una funzione specifica, e che vi sia una netta separazione fra i vari realizzati; tendenzialmente, a ognuno è dedicato un apposito file.

I nomi dei componenti adottano la notazione camelCase, e, in aggiunta, la prima lettera deve essere maiuscola. Una volta scritto il codice, il componente viene esportato grazie alla parola chiave (in inglese *keyword*) `export` di JavaScript (ad esempio, `export default FunctionComponent`). Così facendo, questo viene reso disponibile all'utilizzo all'interno del progetto. Quando il componente deve essere utilizzato all'interno di un file, invece, viene importato tramite la parola chiave `import`, seguita dal suo nome e dalla sua posizione<sup>15</sup> (ad esempio, `import FunctionComponent from './components/FunctionComponent.js'`). Una volta che il componente è stato importato può essere utilizzato richiamando, dove desiderato, la sua istanza; l'istanza di un componente può essere richiamata anche all'interno di un altro componente.

#### 4.1.2.1 *Class vs Function*

I componenti di React possono essere di due tipi, di classe (in inglese *class components*) e di funzione (in inglese *function components*); il nome di entrambi è ispirato al metodo impiegato per la loro creazione.

I primi rappresentano la versione più tradizionale ed elementare di quest'elemento alla base di React. La loro definizione avviene attraverso il concetto, comune nella maggior parte dei linguaggi di programmazione (fra cui anche JavaScript), di classe. Questo costrutto rappresenta una sorta di *template* atto alla creazione di oggetti che condividono delle caratteristiche.

Poiché anche i componenti di classe ne possiedono di comuni, per evitare che una serie di proprietà e metodi vengano riscritti ogni singola volta, la loro creazione avviene tramite l'esten-

---

<sup>15</sup>Solitamente, le importazioni in generale (ad esempio, di componenti, librerie...) vengono effettuate nella parte alta del file.

sione della classe Component attraverso la dichiarazione di classe.

Per questo motivo, come possibile vedere a riga 1 del Listing 4.6, il nome del componente, ClassComponent, è preceduto dalla parola chiave `class` e seguito da `extends`.

Un nuovo componente, quindi, altro non è che una classe figlia che estende la classe Component<sup>16</sup>.

Il corpo del componente di classe, ovvero il contenuto delle parentesi graffe, fornisce delle istruzioni per il *rendering* dell'elemento a schermo. Di vitale importanza è il metodo `render()`, l'unico che deve essere necessariamente presente in ogni componente; all'interno del suo corpo, la parola chiave `return` è seguita da tutto il codice JSX che si desidera venga visualizzato a schermo all'interno dell'interfaccia.

Il contenuto del corpo del metodo `render()` illustrato da riga 2 a 4 è un semplice `<h1>`.

```

1 export class ClassComponent extends React.Component {
2   render() {
3     return <h1>Hello world</h1>;
4   }
5 }
```

Listing 4.6: Componente di classe

I componenti funzione (o funzionali), invece, rappresentano una versione allo stesso tempo più avanzata e semplice di quest'elemento. La loro creazione avviene attraverso la dichiarazione di funzione; per questo motivo, fanno uso della parola chiave `function`, come illustrato dal Listing 4.7, oppure della sintassi funzione freccia<sup>17</sup>. A differenza dei componenti di classe, il loro corpo non presenta il metodo `render()`<sup>18</sup> ma solamente la parola chiave `return` e codice il JSX che andranno a renderizzare.

```

1 export function FunctionComponent() {
2   return <h1>Hello world</h1>;
```

---

<sup>16</sup>La classe Component rappresenta una delle proprietà dell'oggetto React, oggetto disponibile grazie all'importazione dalla libreria react a inizio file.

<sup>17</sup>La sintassi funzione freccia (in inglese *arrow function*) prevede che la funzione venga dichiarata attraverso la parola chiave `const` seguita dal nome della funzione e da un uguale (`=`). Quest'ultimo simbolo è a sua volta seguito dall'argomento della funzione (solo se l'argomento è assente, o se gli argomenti son più di uno è richiesto l'utilizzo delle parentesi tonde `(( ))`) e da una freccia (`=>`) che precede il corpo della funzione, racchiuso dalle parentesi graffe. La sintassi funzione freccia è stata introdotta con ECMAScript 6 (ES6).

<sup>18</sup>Di conseguenza, i componenti di funzione non presentano nemmeno le parentesi graffe del corpo del metodo `render()`.

<sup>3</sup> }

#### Listing 4.7: Componente funzionale

Nonostante a partire dalla versione 16.8.0 di React i componenti di classe e quelli funzionali condividono le stesse funzionalità, è possibile affermare che i secondi siano migliori in termini di pulizia del codice: questi offrono un modo più elegante e conciso di creare componenti. Inoltre, alcuni sviluppatori li preferiscono ai componenti di classe per la loro semplicità d'utilizzo grazie a funzioni come, ad esempio, gli *hooks*<sup>19</sup>. Il codice stesso che si cela dietro alle implementazioni relative al progetto oggetto dell'elaborato utilizza unicamente componenti funzionali per i suddetti motivi.

In definitiva, è bene sottolineare che i componenti funzionali sono facoltativi e retro compatibili: questi possono essere utilizzati congiuntamente con i componenti di classe poiché non costituiscono il loro rimpiazzo, ma solamente una pregiata alternativa.

#### 4.1.2.2 props

I componenti di React sono in grado di relazionarsi fra di loro scambiandosi delle informazioni, meglio conosciute come *props*<sup>20</sup>.

Il passaggio di *props* da componente a componente avviene fornendo all'istanza di un componente un attributo dotato di un valore<sup>21</sup>; il nome dell'attributo può essere di qualsiasi tipo, è a discrezione dello sviluppatore, e ogni volta che deve essere passato un qualcosa che non sia una stringa, il valore deve essere racchiuso fra parentesi graffe<sup>22</sup>.

Nell'esempio del Listing 4.8, come possibile vedere da riga 7 a 10, l'istanza del componente `<NestedComponent />` è dotata delle *props* `example` e `handleClick`. I valori a esse associati sono, rispettivamente, la stringa `example` e la funzione `handleClick`.

<sup>19</sup>Gli *hooks* sono descritti nella sottosezione 4.1.3.

<sup>20</sup>Tutti i componenti sono dotati dell'oggetto *props*, oggetto che contiene le informazioni passate fra componenti, le *props*. Il termine *props*, quindi, ha un duplice utilizzo: viene usato sia per indicare l'oggetto che contiene le informazioni passate, sia per riferirsi al plurale di *prop*, quando queste sono più di una.

<sup>21</sup>Il meccanismo di passaggio delle *props* ricorda molto quello degli attributi in HTML.

<sup>22</sup>Un tipo particolare di valore che si è soliti passare fra componenti sono le funzioni di gestione dell'evento (in inglese *event handlers functions*). Affinché il loro passaggio vada a buon fine, queste devono essere dichiarate all'interno del corpo del componente. La convenzione prevedere che il nome della funzione che verrà assegnata come valore di uno specifico attributo ascoltatore di eventi sia composto dalla parola *handle* seguita dal nome dell'attributo ascoltatore di eventi (ciò che segue l' *on*). Come mostrato a riga 2 del Listing 4.8, la funzione `handleClick` verrà assegnata come valore dell'attributo ascoltatore di eventi `onClick`.

```

1 export function FunctionComponent() {
2   const handleClick = () => {
3     console.log('click done');
4   };
5
6   return (
7     <NestedComponent
8       example="example"
9       handleClick={handleClick}
10    />
11  );
12}

```

Listing 4.8: Passaggio di *props*

Per poter usufruire del valore delle *props* è necessario accedere a esse; il metodo d’accesso differisce a seconda del tipo di componente in uso.

Nei componenti di classe, le informazioni passate fra componenti sono rese disponibili attraverso la sintassi `this.props` seguita da un punto e dal nome della *prop* alla quale si vuole accedere.

Nell’esempio del Listing 4.9, come mostrato alle righe 5 e 6, l’accesso alle informazioni avviene grazie alle espressioni `this.props.example` e `this.props.handleClick`.

```

1 export class NestedComponent extends React.Component {
2   render() {
3     return (
4       <>
5         <h1>{this.props.example}</h1>
6         <button onClick={this.props.handleClick}>Click here!</button>
7       </>
8     );
9   }
10 }

```

Listing 4.9: Accesso alle *props* in un componente di classe

Nei componenti funzionali, invece, l’accesso alle *props* viene garantito passando l’oggetto `props` come argomento del componente stesso, e grazie all’uso della sintassi `props` seguita da un punto e dal nome della *prop* alla quale si vuole accedere.

Nell'esempio del Listing 4.10, il componente funzionale `NestedComponent` è dotato dell'argomento `props` (a riga 1) e l'accesso alle informazioni avviene tramite le espressioni `props.example` e `props.handleClick` (alle righe 4 e 5).

```

1 export function NestedComponent(props) {
2   return (
3     <>
4       <h1>{props.example}</h1>
5       <button onClick={props.handleClick}>Click here!</button>
6     </>
7   );
8 }
```

Listing 4.10: Accesso alle `props` in un componente funzionale

In aggiunta, per una migliore qualità del codice, nei componenti funzionali è possibile accedere alle `props` avvalendosi della destrutturazione dell'oggetto (in inglese *object destructuring*). Questa pratica consiste nell'inserire le `props` alle quali si desidera accedere all'interno delle parentesi graffe separandole con una virgola, per poi accedere a grazie al solo utilizzo del loro nome<sup>23</sup>.

Nell'esempio del Listing 4.11, al componente funzionale `NestedComponent` è passato un argomento che, avvalendosi della destrutturazione dell'oggetto, si compone delle `props` `example` e `handleClick` (riga 1). All'interno del corpo del componente, l'accesso alle `props` avviene semplicemente con l'utilizzo del loro nome (riga 4 e 5).

```

1 export function NestedComponent({ example, handleClick }) {
2   return (
3     <>
4       <h1>{example}</h1>
5       <button onClick={handleClick}>Click here!</button>
6     </>
7   );
8 }
```

Listing 4.11: Accesso con destrutturazione alle `props` in un componente funzionale

---

<sup>23</sup>La destrutturazione dell'oggetto permette di rimuovere dal codice la sintassi `props` seguita dal punto, che in progetti di grandi dimensioni può essere ripetuta numerose volte.

All'interno dei componenti, le *props* vengono quasi sempre utilizzate o per renderizzare dati a schermo, oppure per specificare le funzioni da eseguire in relazione agli attributi ascoltatori di eventi<sup>24</sup>. Quindi, è comune imbattersi nelle sintassi appena descritte nel codice all'interno del corpo del metodo `render()`, nel caso dei componenti di classe, oppure nel codice che segue la parola chiave `return` nel caso dei componenti funzionali. Qui, le *props* vengono richiamate all'interno di tag, oppure per l'assegnazione del valore di un attributo ascoltatore di eventi.

#### 4.1.2.3 Stato

I componenti di React possono accedere a delle informazioni anche attraverso lo stato.

A differenza delle *props*, lo stato non proviene dall'esterno: questo viene definito internamente al componente, il quale viene dotato della proprietà `state`. Come mostrato da riga 2 a 5 del Listing 4.12, la proprietà `state` (all'interno della sintassi `this.state`) si trova sempre all'interno di un metodo costruttore, `constructor(props)`, il quale dev'essere necessariamente dotato del metodo `super(props)`.

Il valore di `this.state` dev'essere un oggetto che rappresenta lo "stato" iniziale di qualsiasi istanza del componente.

In un componente di classe, similmente a quanto visto per le *props*, l'accesso a una proprietà contenuta nello stato viene effettuato attraverso la sintassi `this.state` seguita da un punto e dal nome della proprietà alla quale si vuole accedere.

Nell'esempio del Listing 4.12, come possibile vedere a riga 14, l'accesso alla proprietà contenuta nello stato è effettuato grazie alla sintassi `this.state.example`.

Inoltre, lo stato è caratterizzato dalla possibilità di essere modificato: all'interno di un componente, il suo valore iniziale può essere aggiornato grazie all'invocazione della funzione `this.setState()`. Seppur questa accetti due argomenti, quello più importante e comunemente utilizzato è l'oggetto, ovvero ciò che andrà a sostituire lo stato iniziale precedentemente dichiarato dentro costruttore<sup>25</sup>.

Sempre nello stesso esempio, come illustrato da riga 7 a 9, la funzione `this.setState()`, a cui è passato come argomento l'oggetto `example`: '`example2`', è stata utilizzata all'interno del corpo della funzione `handleClick()`.

---

<sup>24</sup>Gli attributi ascoltatori di eventi di JSX sono descritti nella sottosezione 4.1.1.

<sup>25</sup>Il secondo argomento accettato dalla funzione `this.setState()` è una funzione di *callback*. Come già detto, tale argomento non viene quasi mai usato.

```

1 export class ClassComponent extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = { example: 'example1' };
5   }
6
7   handleClick() {
8     this.setState({ example: 'example2' });
9   }
10
11  render() {
12    return (
13      <>
14        <h1>{this.state.example}</h1>
15        <button onClick={() => this.handleClick()}>Click here!</button>
16      </>
17    );
18  }
19}

```

Listing 4.12: Dichiarazione e aggiornamento dello stato in un componente di classe

All'interno dei componenti funzionali, invece, la gestione dello stato avviene in maniera più semplice e immediata tramite l'*hook* `useState()`<sup>26</sup>.

### 4.1.3 Hooks

Dal suo rilascio nel 2013, React è cambiato molto nel corso degli anni. Senz'ombra di dubbio, una delle introduzioni più importanti è stata apportata dalla versione 16.8.0 della libreria, la quale è riuscita a svecchiare e semplificare ulteriormente il funzionamento grazie all'introduzione degli *hooks*.

La documentazione definisce l'*hook* come:

«[...] una speciale funzione che ti permette di “agganciare” funzionalità di React»[20].

Gli *hooks* «[...] permettono di utilizzare più funzioni di React senza dover ricorrere alle classi»[14], quindi, anche con i componenti funzionali. Per la precisione, gli *hooks* funzionano solamente in combinazione con questi ultimi; assieme a essi contribuiscono alla scrittura di un

---

<sup>26</sup>L'*hook* `useState()` è descritto nella sottosottosezione 4.1.3.1.

codice più leggibile e intuitivo.

Ne esistono di due tipi: *built-in*, ovvero quegli *hooks* presenti nella libreria React a priori, oppure personalizzati, ovvero quegli *hooks* realizzati appositamente dagli sviluppatori per soddisfare delle esigenze su misura. Delle convenzioni specificano come deve essere assegnato il nome a un *hook* personalizzato in fase di creazione: deve adottare la notazione camelCase e, più nello specifico, utilizzare la parola `use` seguita da un'altra, scelta a piacimento, solitamente connessa del compito assolto dall'*hook* (ad esempio, `useExample()`).

Gli *hooks* possono essere invocati all'interno degli *hooks* personalizzati ma non «[...] all'interno di cicli, condizioni o funzioni nidificate»[16].

Proprio come i componenti funzionali<sup>27</sup>, queste particolari funzioni sono facoltative e retro compatibili. Infatti, è possibile «provare [...] gli *hooks* in pochi componenti senza dover riscrivere alcun codice esistente [...]» e senza obblighi, e il loro impiego non comporta «[...] alcun cambiamento che possa rompere funzionalità esistenti»[15].

In ogni caso, però, l'utilizzo degli *hooks* è vincolato alla loro importazione all'interno di un progetto; ancora una volta, è necessario utilizzare la parola chiave `import` seguita dal nome dell'*hook*, seguito dalla parola chiave `from`, e da '`react`' nel caso di un *built-in hook*, oppure dalla posizione nel caso di un *hook* personalizzato (ad esempio, '`./hooks/useExample`').

#### 4.1.3.1 *Built-in hooks*

I più importanti *built-in hooks* di React utilizzati all'interno del progetto oggetto dell'elaborato sono:

- `useState()`: permette di dichiarare una variabile di stato, fornendo un «[...] modo per “conservare” qualche valore [...]»[21] e andando a soppiantare la sintassi `this.state` tipica dei componenti di classe: permette di «[...] aggiungere lo state React nei componenti funzione»[21].

Quest'*hook* accetta un solo argomento, un qualsiasi valore che consente di inizializzare lo stato iniziale. I valori che `useState()` restituisce sono lo stato corrente, e una funzione per il suo aggiornamento, il *setter*.

Nell'esempio del Listing 4.13, lo stato corrente è rappresentato dalla variabile di stato

<sup>27</sup>Si veda la sottosottosezione 4.1.2.1.

`example`, il cui valore iniziale è la stringa '`example`', mentre il *setter* è costituito dalla funzione `setExample()`<sup>28</sup>.

```
1 const [example, setExample] = useState('example');
```

Listing 4.13: *Built-in hook useState()*

- `useEffect()`: permette di eseguire una funzione dopo che un componente è stato rende- rizzato: «[...] React ricorderà la funzione passata [...] e la invocherà dopo aver eseguito gli aggiornamenti del DOM»[18].

Quest'*hook* accetta due argomenti: una funzione, ovvero ciò che deve essere ricordato ed eseguito, e un'*array* di dipendenze<sup>29</sup>. Se non viene specificato un secondo argomento, l'*hook* prevede che la funzione (qui anche chiamata “effetto”) venga eseguita sia dopo il primo *rendering* che a ogni aggiornamento del componente.

Alcuni effetti potrebbero richiedere che venga effettuata della pulizia; passando un *array* vuoto (`[]`) l’effetto viene eseguito e pulito una sola volta perché questo «[...] non dipende da alcun valore di oggetti di scena o di stato, quindi non ha bisogno di essere rieseguito»[19].

Posizionando l’`useEffect()` all’interno di un componente, si garantisce all’*hook* l’acces- so ad elementi come, ad esempio, *stati* o *props* presenti nel componente stesso.

Nell’esempio del Listing 4.14, la funzione `console.log()` con argomento `example` vie- ne eseguita ogni volta che avviene un cambiamento della dipendenza specificata, ovvero la variabile `example`.

```
1 useEffect(() => {console.log(example)}, [example])
```

Listing 4.14: *Built-in hook useEffect()*

- `useContext()`: permette di gestire lo stato globalmente evitando il cosiddetto *props drilling*, il passaggio di «[...] *props* [...] a componenti annidati, attraverso componenti che non ne hanno bisogno»[4].

Quest'*hook* accetta un solo argomento, l’oggetto contesto, ovvero il valore restituito dal metodo `createContext()`. L’unico valore restituito da `useContext()` è quello corrente

---

<sup>28</sup>Nei componeneti di classe, avremmo trovato le sintassi `this.state.example` e `this.setExample()`.

<sup>29</sup>All’interno degli *hooks*, le dipendenze (in inglese *dependencies*) sono dei valori al variare di quali viene eseguita nuovamente una funzione.

del contesto; tale valore è determinato da quello assegnato alla *prop value* del componente `<ExampleContext.Provider>`.

Nell'esempio del Listing 4.15, alla variabile `example` viene associato il valore recuperato attraverso l'argomento `ExampleContext`, ovvero il valore della *prop value*.

```
1 const example = useContext(ExampleContext)
```

Listing 4.15: *Built-in hook* `useContext()`

- `useReducer()`: permette di gestire una logica di stato personalizzata, per esempio, articolata in sotto valori, oppure in cui vi è un'influenza di uno stato su un altro; rappresenta «un'alternativa allo `useState()`»[12].

Quest'*hook* accetta come argomenti un riduttore, la cui sintassi è `(state, action) => newState`, ed uno stato iniziale<sup>30</sup>. I valori che `useReducer()` restituisce sono lo stato corrente ed un metodo di invio.

Nell'esempio del Listing 4.16, lo stato corrente è rappresentato dalla variabile di stato `example` ed il metodo di invio dalla funzione `dispatch()`. Il riduttore è invece rappresentato dalla funzione `reducer()` la quale opera sull'oggetto `initialExamples`.

```
1 const [examples, dispatch] = useReducer(reducer, initialExamples);
```

Listing 4.16: *Built-in hook* `useReducer()`

- `useMemo()`: permette di ritornare un valore memorizzato evitando che funzioni dispendiose a livello computazionale vengano eseguite ad ogni *rendering* del componente; ottimizza le *performance* generali.

Quest'*hook* accetta come argomenti una funzione, ovvero ciò che potrebbe causare rallentamenti, e un'*array* di dipendenze<sup>31</sup>.

Nell'esempio del Listing 4.17, la funzione `functionExample(a, b)` viene eseguita solamente quando avviene un cambiamento delle sue dipendenze, `a` e `b`, e il valore della dispendiosa computazione da essa effettuata viene assegnato alla variabile `example`.

```
1 const example = useMemo(() => functionExample(a, b), [a, b]);
```

Listing 4.17: *Built-in hook* `useMemo()`

---

<sup>30</sup>Lo stato iniziale, il secondo argomento, solitamente contiene un oggetto.

<sup>31</sup>Si veda la nota 29.

- `useRef()`: permette di archiviare un valore rendendolo persistente attraverso le renderizzazioni.

Quest’*hook* accetta un solo argomento, un qualsiasi valore che consente di inizializzare la proprietà `.current`. Quest’ultima appartiene all’oggetto `ref`, l’unico valore restituito dallo `useRef()`, valore che persistrà durante l’intero ciclo di vita del componente.

Nell’esempio del Listing 4.18, la proprietà `.current` dell’oggetto `modalExample` è inizializzata con la stringa `'example'`.

```
1 const refExample = useRef('example');
```

Listing 4.18: *Built-in hook useRef()*

#### 4.1.3.2 Hooks personalizzati

Gli *hooks* personalizzati (in inglese *custom hooks*) realizzati per fornire una soluzione a problematiche specifiche utilizzati all’interno delle implementazioni oggetto dell’elaborato sono:

- `useApi()`: permette di effettuare una chiamata ad un’API<sup>32</sup> per il recupero di dati.

Quest’*hook* accetta tre argomenti: la funzione che effettua la chiamata all’API, dei parametri e un valore iniziale. I valori restituiti da `useApi()` ed utilizzati sono: un’*array* di oggetti, delle informazioni sullo stato della chiamata effettuata, ed una funzione per aggiornare l’*array* di oggetti reperendo di nuovo i dati.

La struttura interna dell’*hook*<sup>33</sup>, seppur includa anche lo `useState()` e lo `useEffect()`, è basata sul *built-in hook* `useReducer()`.

Nell’esempio del Listing 4.19, con l’utilizzo della funzione `getExamples()`, viene effettuata una chiamata ad un’API e vengono recuperate le seguenti proprietà: i dati contenuti nell’*array* `examples`, le informazioni di stato delle variabili `isLoading` ed `error`, e la funzione per l’aggiornamento dei dati `fetchExamples()`<sup>34</sup>.

```
1 const {
2   data: examples,
3   isLoading,
```

---

<sup>32</sup>Si veda la sottosottosezione 4.3.1.1

<sup>33</sup>Rimando a nota che fornisce un esempio di utilizzo dell’`useReducer()`.

<sup>34</sup>Generalmente, la funzione per l’aggiornamento dei dati viene invocata una volta effettuata un’operazione, ad esempio, dopo l’aggiunta di un elemento utilizzando una finestra di dialogo.

```

4   error ,
5   silent: fetchExamples ,
6 } = useApi(getExamples, [], []);

```

Listing 4.19: Cutom hook useApi()

- `useModal()`: permette di reperire una serie di funzioni e variabili necessari al funzionamento delle finestre di dialogo (anche dette “modali”).

Quest’*hook* accetta un solo argomento, un oggetto di opzioni per l’impostazione dei dati; i valori restituiti da `useModal()` ed utilizzati sono: le funzioni `open()` e `close()`, per l’apertura e la chiusura della modale, e gli stati `isOpen` e `data`, e il *setter* `setData()` per la gestione dello stato di apertura della modale, e dei dati su cui essa operare.

La struttura interna dell’*hook* si avvale dell’utilizzo del *built-in hook* `useState()`. Nell’esempio del Listing 4.20, l’oggetto `modalExample` “importa”, sottoforma di proprietà, tutti gli elementi discussi appena sopra attraverso l’uso dell’*hook* `useModal`.

```

1 const modalExample = useModal();

```

Listing 4.20: Cutom hook useModal()

- `useValidation()`: permette di effettuare delle validazioni.

Quest’*hook* accetta solamente un argomento, l’oggetto `schema`, oggetto le cui proprietà specificano le caratteristiche dei valori che devono essere validati. I valori restituiti da `useValidation()` ed utilizzati sono delle funzioni che permettono di eseguire diversi tipi di validazioni.

La struttura interna dell’*hook* si avvale dell’utilizzo del *built-in hook* `useState()`. Nell’esempio del Listing 4.21, viene effettuata la validazione dell’oggetto `schema`<sup>35</sup> e vengono recuperate proprietà `validate`, `validationProps` e `validateField` per la validazione di valori specifici.

```

1 const { validate, validationProps, validateField } =
2 useValidation(schema);

```

Listing 4.21: Custom hook useValidation()

---

<sup>35</sup>L’*hook* personalizzato `useValidation()`, effettuando validazioni sull’oggetto `schema`, è utilizzato in combinazione alla libreria *Yup*, la quale verrà trattata nella sottosezione 4.3.3.

- `useStatus()`: permette di gestire lo stato all'interno della logica di una funzione *event handler*.

Quest'*hook* accetta un'argomento, uno stato iniziale che verrà utilizzato dal riduttore dello `useReducer()`, il *built-in hook* alla base della struttura del *custom* in analisi. I valori restituiti da `useStatus()` e utilizzati sono le funzioni `setLoading()`, `setSuccess()` e `setError()`, delle funzioni che gestiscono i casi in cui l'*handler*, rispettivamente, sta venendo eseguito, è stato eseguito con esito positivo, ed è stato eseguito con esito negativo. Nell'esempio del Listing 4.22, l'oggetto `statusExample` "importa", sottoforma di proprietà, gli elementi discussi appena sopra attraverso l'uso dell'*hook* `useStatus()`.

```
1 const statusExample = useStatus();
```

Listing 4.22: *Custom hook useStatus()*

## 4.2 Material UI

Material UI (abbreviato in MUI) è una libreria per la creazione di interfacce utente che contiene diversi componenti React *open-source* che implementano il *Material Design* di Google.

L'obiettivo che la libreria si pone è quello di semplificare e velocizzare il lavoro degli sviluppatori *front-end* nella realizzazione di esperienze utente intuitive, gradevoli e uniformi su tutti i tipi di dispositivi, mantendendo l'elevato standard qualitativo di *Material Design*. Per raggiungere questo traguardo, Material UI mette a disposizione dello sviluppatore una grande varietà di componenti già realizzati, i quali possono essere implementati all'interno di un progetto immediatamente.

Le principali caratteristiche che hanno determinato il successo di questa libreria sono:

- **Semplicità:** MUI offre un'esperienza di sviluppo intuitiva, capace di facilitare l'approccio agli sviluppatori inesperti da un lato, e di contribuire positivamente alla coesione generale del team dall'altro;
- **Affidabilità:** MUI rappresenta la libreria per lo sviluppo di UI pensate appositamente per React con la più grande community. La sua presenza "in scena" a partire dalla comparsa di React stesso e il grande numero di persone, fra cui anche il team di designer e di sviluppatori di Google, che si occupano del suo mantenimento quotidianamente sono i fattori le

conferiscono una grande affidabilità. Material UI è utilizzato da numerose aziende di tutto il mondo;

- **Velocità:** data la vastità della community che si è dedicata alla cura dei componenti della libreria, utilizzando MUI, un team è in grado di focalizzarsi maggiormente sul funzionamento dell'applicativo in sviluppo, quasi dimenticandosi dell'interfaccia grafica. La UI, potenzialmente, può essere "presa a carico" per intero da Material UI riducendo estremamente i tempi di realizzazione di un applicativo;
- **Estetica:** implementando fedelmente *Material Design*, MUI è sia dotato di un'estetica gradevole, che di componenti ottimamente progettati in termini di forma e di funzione. Nonostante ciò, la libreria è in grado di discostarsi da *Material Design* ogni qualvolta sia necessario fornire più soluzioni;
- **Personalizzazione:** MUI è personalizzabile in modo intuitivo; sul sito ufficiale della libreria sono mostrati vari *template* che illustrano quanto la libreria possa essere customizzata.

### 4.2.1 Material Design

In quanto libreria, Material UI permette di costruire interfacce utente di estrema qualità perché basata su *Material Design*, il *design system* realizzato e mantenuto da Google nato per l'implementazione di porzioni di interfacce utente su sistemi operativi, come *Android* e *Flutter*, e su web. Ciò rende MUI veicolo di un linguaggio grafico diffuso in maniera capillare e conosciuto in tutto il mondo.

*Material Design* è stato annunciato da Google nel 2014 con l'intenzione di uniformare l'esperienza utente di tutti i prodotti offerti dalla *software house*, e di dar vita a uno strumento visivo rivoluzionario, dotato di una proposta stilistica più concreta e materiale. Il termine “*Material*”, infatti, altro non è che una metafora riferita alla più peculiare caratteristica di questo *design system*, ovvero quella di ispirarsi a veri materiali presenti nel mondo reale e alle loro proprietà fisiche. In sostanza, ognuno dei suoi elementi grafici, «[...] dovrebbe essere trattato come un oggetto fisico con una fisicità propria»[22].

M. Duarte, UI designer nonché vice presidente del reparto *design* di Google, affermò che:

«A differenza della carta reale, il nostro materiale digitale può espandersi e riformarsi in modo intelligente. *Material Design* è dotato di superfici fisiche e bordi. Cuciture e ombre conferiscono significato a ciò che stai toccando».

All’intero delle interfacce grafiche realizzate con *Material Design*, di primaria importanza è la terza dimensione, l’elemento che permette all’utente di percepire profondità e struttura dei componenti, e di star interagendo con qualcosa di tangibile.

### 4.2.2 Componenti

Quando si parla di sviluppo di applicazioni web e mobile, Material UI rappresenta un punto di riferimento in virtù della sua semplicità d’utilizzo.

La libreria adotta un approccio modulare fornendo una vasta gamma di componenti, delle porzioni di interfaccia grafica riutilizzabili, retrocompatibili e personalizzabili, già sviluppate e mantenute, facili da integrare all’interno di un progetto esistente e disponibili fin da subito.

I componenti di MUI sono raggruppati in categorie a seconda della funzione che assolvono; mantenendo l’ordine della documentazione ufficiale, di seguito ne verrà offerta una panoramica, e saranno analizzati i componenti e le loro *props* assegnategli<sup>36</sup> all’interno delle implementazioni.

- **Inputs:** consentono all’utente di effettuare azioni o di compiere scelte all’interno di una gamma di opzioni.

Componenti usati:

- Button e IconButton: permettono la creazione di bottoni che consentono all’utente di effettuare delle scelte. Mentre Button può contenere (fra i tag di apertura e chiusura) solamente del testo, IconButton è in grado di contenere anche un’icona.

*Props* utilizzate: size (stabilisce la dimensione del componente, ad esempio, small), startIcon (permette di associare un’icona anche al componente Button);

- Checkbox: permette la creazione di caselle di controllo che consentono all’utente di selezionare uno o più elementi (da una serie), oppure spuntare un’opzione come attiva o disattiva.

*Props* utilizzate: checked (stabilisce se il componente è *flaggato*) e onChange (stabilisce la funzione eseguita quando lo stato valore cambia);

- FormGroup: permette la creazione di caselle di controllo con una struttura personalizzata, ad esempio, attraverso il componente FormControlLabel.

---

<sup>36</sup>Poiché alcune *props* hanno il medesimo funzionamento in diversi componenti, la spiegazione della loro funzione viene fornita solamente la prima volta che ci si imbatte in esse.

*Props* utilizzate: `control` (stabilisce il componente di controllo, ad esempio, `CheckBox`) e `label` (definisce l'etichetta, ciò che viene visualizzato all'apice del componente una volta posizionato il cursore sul testo) con `FormControlLabel`;

- `TextField`: permette la creazione di caselle di testo che consentono all'utente di manipolare (inserire o modificare) del testo.

*Props* utilizzate: `fullWidth` (stabilisce che il componente occupi tutto il div dal quale è contenuto in larghezza), `size`, `variant` (definisce lo stile del componente, ad esempio, `filled`), `label`, `placeholder` (definisce il segnaposto, ciò che viene visualizzato all'interno del componente quando il cursore non è ancora posizionato e il valore non è presente), `value` (definisce il valore del testo del componente), `onChange` e `InputProps` (definisce le *props* passate al componente `Input`);

- **Data display:** consentono all'utente di visualizzare dati e/o informazioni strutturate, dotate di organizzazione e formattazione.

Componenti usati:

- `Divider`: permette la creazione di sottili linee per il raggruppamento di elementi; funziona come il tag `<hr>` in HTML;
- *Material Icons*: permettono di arricchire l'esperienza utente inserendo icone che illuminino altri componenti, suggerendo ed esemplificando le loro funzioni all'utente. `AddIcon`, `EditIcon` e `DeleteIcon`, ad esempio, sono alcune fra le principali implementate fra le innumerevoli messe a disposizione direttamente dal sito ufficiale di Material Icons.

*Props* utilizzate: `fontSize` (stabilisce la dimensione del componente, ad esempio, `inherit`);

- `Table`: permette la creazione di tabelle per la visualizzazione di dati. La struttura di una tabella viene realizzata attraverso una gerarchia di specifici componenti<sup>37</sup> come, ad esempio, `TableContainer`, `TableHead`, `TableBody`, `TableRow` e `TableCell`.

*Props* utilizzate: `size` con `Table`, `component` (stabilisce il componente usato per il nodo radice) con `TableContainer`, e `align` (stabilisce l'allineamento del compone-

<sup>37</sup>La creazione di una tabella prevede che venga rispettata una specifica gerarchia di tag anche in HTML.

te, ad esempio, `right`) e `colSpan` (stabilisce quante colonne una cella deve coprire) con `TableCell`;

- `Typography`: permette il *rendering* di testi statici.  
`Props` utilizzate: `variant` (definisce il *font* del componente, ad esempio, `title1...`) e `size`;
- `FormHelperText`: permette la creazione di messaggi di suggerimento che suggeriscono all’utente come svolgere un’azione;

- ***Feedback***: consentono all’utente di ricevere un ricoscontro, ad esempio, a seguito di un’interazione con un componente di tipo *input*.

Componenti usati:

- `Alert`: permette la creazione di brevi messaggi informativi per l’utente;
- `Dialog`: permette la creazione di finestre di dialogo (anche dette modali) che informano l’utente su un compito particolare: queste possono contenere informazioni critiche, richiedere decisioni o coinvolgere più compiti. Anche la struttura di una finestra di dialogo viene realizzata attraverso una gerarchia di specifici componenti<sup>38</sup> come, ad esempio, `DialogTitle`, `DialogActions` e `DialogContent`.

`Props` utilizzate: `open` (quando il suo valore è `true`, stabilisce che il componente deve essere mostrato), `onClose` (stabilisce la funzione eseguita quando il componente deve essere chiuso), `scroll` (stabilisce il contenitore per effettuare lo *scrolling* del componente), `fullWidth` e `maxWidth` (stabilisce la larghezza massima del componente) con `Dialog`;

- ***Surfaces***: offrono superfici su cui visualizzare dati e/o informazioni, come, ad esempio, quelle strutturate grazie ai componenti di tipo *data display*.

Componenti usati:

- `Card`: permette la creazione di “perimetri” tematici dotati di contenuti e di azioni; questi consentono all’utente di comprendere rapidamente la strutturazione di più alto livello degli elementi. Analogamente a quanto visto per `Table` e `Dialog`, anche nel caso del componente `Card` è prevista una struttura gerarchica di componenti<sup>39</sup> come,

---

<sup>38</sup>Si veda nota 37.

<sup>39</sup>Si veda nota 37.

ad esempio, `CardHeader`, `CardAction`, `CardContent` e `CardMedia`.

*Props* utilizzate: `subheader` (definisce il sottotitolo) e `action` (definisce le azioni che possono essere svolte) con `CardHeader`, e `component` (stabilisce il componente usato per il nodo radice) e `image` (stabilisce l'immagine da visualizzare) con `CardMedia`;

- `Paper`: permette la creazione di componenti che riproducono l'effetto della carta;

- ***Navigation***: consentono all'utente di navigare agevolmente, offrendo punti di riferimento e scorciatoie;

- ***Layout***: consentono all'utente di visualizzare i dati e/o le informazioni strutturate in modo coerente e ordinato all'interno della pagina.

Componenti usati:

- `Box`: permette la creazione di elementi “container” per la gestione del CSS;

- `Grid`: permette la creazione di una griglia per l'organizzazione dei contenuti.

*Props* utilizzate: `container` (stabilisce che il componente ha lo stesso comportamento di un contenitore a cui è attribuita la proprietà `flex`), `spacing` (stabilisce la spaziatura fra i diversi componenti di tipo `item`; funziona solamente componenti di tipo `container`), `item` (stabilisce che il componente è di tipo `item`, quindi figlio del componente di tipo `container`), `xs/md/lg` (quando il loro valore è un numero<sup>40</sup>, stabilisce la spaziatura fra i componenti di tipo `item`; a seconda di quale delle tre *props* viene fornita, il valore numerico viene applicato ai *breakpoint* `xs`, `md` o `lg`);

- ***Utils***: offrono allo sviluppatore delle soluzioni semplici ed efficaci per la modellazione dell'interfaccia. Componenti usati:

- `Collapse`: fa parte delle transazioni e consente di espandere un componente.

*Props* utilizzate: `in` (stabilisce l'espansione o la chiusura) e `timeout` (stabilisce la durata della transazione);

- ***MUI X***: offrono allo sviluppatore una serie di componenti avanzati per la modellazione di casistiche più complesse e specifiche.

Componenti usati:

---

<sup>40</sup>Il valore numerico associato alle *prop* `xs/md/lg` non può mai essere superiore a 12, ovvero il numero totale di colonne presenti all'interno di un contenitore.

- **DatePicker**: permette la creazione di un campo all'interno del quale l'utente può selezionare un valore fra quelli forniti in modo predefinito, in questo caso una data.  
*Props* utilizzate: `label`, `value`, `onChange` e `renderInput` (stabilisce come personalizzare il campo di immissione specificando le *props* del componente `TextField` che devono essere passate).

## 4.3 Altre librerie

Il progetto oggetto dell'elaborato ha visto anche il coinvolgimento di una serie di librerie "secondarie" adibite all'implementazione di funzionalità e alla gestione di problemi specifici. Quelle utilizzate nel corso delle implementazioni sono tratte di seguito.

### 4.3.1 Dialogo client-server

La gestione delle dinamiche relative allo scambio bidirezionale di dati all'interno del binomio client-server rappresenta una tematica che riveste un ruolo di vitale importanza per ciò che concerne il funzionamento di un applicativo. Infatti, il compito del server non è unicamente quello di ospitare la UI per inviarla al browser ogni volta che viene richiesta una pagina lato client, ma anche di archiviare i dati necessari affinché l'applicativo funzioni in modo corretto.

Il passaggio di informazioni fra client e server avviene attraverso richieste effettuate da *Application Programming Interface* (API) all'interagire dell'utente, lato client, con specifiche porzioni dell'interfaccia grafica dell'applicativo (ad esempio, quando viene effettuato un click su un bottone).

#### 4.3.1.1 API REST

Al centro dello scambio di informazioni vi è l'API, l'interfaccia di programmazione delle applicazioni, elemento definito come:

«[...] un insieme di definizioni e protocolli con i quali vengono realizzati e integrati *software applicativi*»[28].

In generale, le API consentono a un applicativo di relazionarsi con l'esterno, interfacciandosi con altri applicativi, dispositivi o servizi.

L'applicativo oggetto dell'elaborato opera con le API REST (anche dette API RESTful), le quali prendono il nome dallo stile architettonico che implementano, ovvero *Representational*

### *State Transfer (REST).*

Le API REST sono state definite per la prima volta nella tesi di dottorato di Roy Fielding, e si sono distinte fin da subito per il grande livello di flessibilità che sono state in grado di offrire agli sviluppatori, specialmente se comparate con altri tipi di API, come, ad esempio, quelle *Single Object Access Protocol* (SOAP). Il loro successo e la loro grande diffusione sono da attribuire all’assenza di vincoli sia rispetto al linguaggio di programmazione usato e al formato di dati accettato, sia alla loro architettura fondata su «[...] sei linee guida che sono risultate più semplici da seguire rispetto a un protocollo prescritto»[29];

Le API REST si avvalgono di richieste HTTP per lo scambio di informazioni; attraverso esse sono in grado di eseguire le operazioni *Create Read Update Delete* (CRUD) su una risorsa:

- GET: permette di recuperare dei dati<sup>41</sup>;
- POST: permette di inviare dei dati;
- PUT: permette di modificare dei dati;
- DELETE: permette di eliminare dei dati.

Le funzioni asincrone che implementano i quattro tipi di chiamata HTTP sono illustrate nel Listing 4.23. Ognuna di esse è dotata degli argomenti url, body e option, tranne quella relativa al metodo GET, la quale, non occupandosi della manipolazione di dati, non necessita dell’argomento body. Tutte quante, inoltre, contengono al loro interno la funzione request(), funzione che concretamente si occupa di eseguire le richiste HTTP attraverso la libreria Axios, e che verrà analizzata nella sottosottosezione 4.3.1.2, la seguente.

```

1 export const get = async (url, options) =>
2   request("GET", url, undefined, options);
3
4 export const post = async (url, body, options) =>
5   request("POST", url, body, options);
6
7 export const put = async (url, body, options) =>
8   request("PUT", url, body, options);
9
10 export const del = async (url, body, options) =>

```

---

<sup>41</sup>Nella maggior parte dei casi, si tratta di *array* di oggetti.

```
11     request("DELETE", url, body, options);
```

Listing 4.23: Funzioni `get()`, `post()`, `put()` e `delete()`

### 4.3.1.2 Axios

La gestione delle richieste HTTP in React è molto comunemente effettuata attraverso la libreria *Axios*, strumento definito dalla documentazione ufficiale come «[...] client HTTP basato su *promise* per node.js e il browser»[3].

La sua caratteristica principale è l’isomorfismo: Axios è in grado di funzionare sia lato client, nel browser, che lato server, in un caso effettuando delle richieste *XMLHttpRequests*, nell’altro con il modulo nativo `http` di node.js.

L’API di *Axios* mette a disposizione una grande varietà di metodi per effettuare richieste HTTP di vario tipo, ad esempio, `axios.request(config)`, `axios.get(url[, config])`, `axios.delete(url[, config])` etc...

All’interno del progetto oggetto dell’elaborato, l’unico utilizzato è `axios()`, funzione che accetta un solo argomento, un oggetto `config` che contiene tutte le specifiche relative alla richiesta da effettuare. I valori ritornati dal metodo sono le proprietà contenute dall’oggetto `config`.

Come possibile vedere nel Listing 4.24, sia il metodo `axios()` che l’oggetto `config` sono contenuti all’interno della funzione freccia asincrona `request()` dichiarata a riga 1. Gli argomenti accettati sono `method`, `path`, `body` e `...options = {}`.

La funzione `axios()`, illustrata a riga 15, è assegnata alla variabile `response`: l’oggetto `config` che gli viene passato come argomento contiene le proprietà mostrate da riga da riga 7 a 13, ovvero `url`, `method`, `headers`, `data: body` e `...options`<sup>42</sup>.

```
1 const request = async (method, path, body, options = {}) => {
2   // [...]
3   const headers = { 'Content-Type': 'application/json' };
4   const baseUrl = api[options.baseUrl || 'chc'];
5   const url = `${baseUrl}/${env}${path}`;
6
7   const config = {
```

<sup>42</sup>Alcune delle proprietà che compongono l’oggetto `config` provengono dagli argomenti passati alla funzione `request`, altre sono dichiarate come variabili all’interno del corpo della funzione stessa.

```

8     url ,
9     method ,
10    headers ,
11    data: body ,
12    ...options ,
13  };
14
15  const response = await axios(config);
16  return response.data;
17 }

```

Listing 4.24: Variabile `request` e utilizzo di `axios(config)`

### 4.3.2 *React Router*

La gestione della navigazione attraverso le differenti pagine che formano l'applicativo costituisce un altro argomento di grande rilievo in fase di sviluppo. Il principale pro delle SPA<sup>43</sup>, ovvero la loro capacità di eseguire tutto il codice della piattaforma all'interno della medesima pagina, rappresenta un potenziale problema. Infatti, poiché la tradizionale navigazione ha abituato a muoversi su più pagine, nell'approcciarsi a una SPA, gli utenti potrebbero sentirsi privati di una serie “punti di riferimento” che forniscono informazioni sulla posizione all'interno di un sito<sup>44</sup>.

Poiché il meccanismo di navigazione predefinito dei browser non c’è, all'interno delle SPA realizzate con React, la gestione degli *Uniform Resource Locator* (URL) avviene manualmente attraverso l'implementazione del *routing*. Solitamente, questa dinamica viene realizzata grazie alla libreria *React Router*, la quale fornisce «[...] un insieme di componenti di navigazione che si compongono in modo dichiarativo [...]»[30] con quelli del progetto in cui vengono integrati.

I componenti che permettono di realizzare il *routing* sono:

- **Router**: contiene il componente responsabile del *rendering* dell'applicazione stessa;

---

<sup>43</sup>Si veda la sottosezione 2.3.2

<sup>44</sup>Fra queste informazioni vi sono, ad esempio, la pagina visitata e la profondità a cui essa si trova all'interno della gerarchia del sito.

- **Switch:** contiene i componenti che appartengono allo stesso URL, ad esempio, gli elementi `Route`<sup>45</sup>;
- **Route:** contiene i componenti delle singole pagine. *Props* utilizzate: `path` (stabilisce la stringa che deve essere aggiunta all'URL);
- **Link:** permette all'utente di navigare su una pagina effettuando un click.

La libreria mette a disposizione dello sviluppatore anche una serie di *hooks* utili per facilitare la gestione della navigazione, come, ad esempio, `useHistory()` (fornisce l'accesso alla cronologia), `useLocation()` (restituisce un oggetto contenente l'URL attuale) e `useParams()` (restituisce un oggetto di coppie chiave/valore dei parametri URL).

### 4.3.3 *Yup*

*Yup* è una libreria per l'analisi e la validazione di valori in JavaScript. La documentazione ufficiale riferisce che è possibile «[...] definire uno schema, trasformare un valore per farlo corrispondere, convalidare la forma di un valore esistente o entrambe le cose»[11].

Il funzionamento di *Yup* prevede la dichiarazione di un oggetto `schema`<sup>46</sup> all'interno del quale, attraverso i metodi forniti, vengono elencati dettagliatamente i vincoli specifici che devono essere rispettati da ogni valore. Ciò è necessario per una loro efficace analisi o validazione. Inoltre, gli schemi realizzati con *Yup* sono intuitivi e permettono di modellare oggetti di tutti i livelli di complessità.

Nell'esempio del Listing 4.25, come mostrato da riga 2 a 5, le proprietà dell'oggetto `schema` sono `example1`, `example2`, `example3` ed `example4`. Le caratteristiche che i loro valori devono rispettare sono, rispettivamente, essere una stringa obbligatoria; essere un valore booleano; essere un'array obbligatorio che contiene almeno un elemento, e in caso contrario restituire un suggerimento; essere una data e restituire un messaggio di errore quando ne viene commesso uno.

```
1 const schema = yup.object().shape({
2   example1: yup.string().required(),
```

---

<sup>45</sup>Per un corretto funzionamento del *routing*, i componenti `Route` devono essere inseriti dal più specifico al più generico.

<sup>46</sup>L'oggetto `schema` viene utilizzato per effettuare delle validazioni grazie all'utilizzo dell'*hook* personalizzato `useValidation()`. Si veda il relativo paragrafo all'interno della sottosottosezione 4.1.3.2.

```

3   example2: yup.boolean(),
4   example3: yup.array().min(1, 'Selezionare almeno un valore').required()
5   example4: yup.date().typeError('Data non valida'),
6 });

```

Listing 4.25: Oggetto schema

### 4.3.4 date-fns

*date-fns* è una libreria per la manipolazione di date in JavaScript (sia lato client, nel browser, che lato server, operando con Node.js) che fornisce un’ampia gamma di strumenti, rimanendo però semplice e consistente.

Alcuni dei pregi che la rendono apprezzata dagli sviluppatori sono la modularità (è possibile utilizzare solamente l’esatta funzione di cui si ha bisogno), la velocità (è piccola e veloce) e il supporto (è ben documentata)[25].

Nell’esempio riportato dal Listing 4.26, la variabile `example` contiene come valore quello della variabile `exampleData`, ma formattato attraverso il metodo `format()` secondo il formato giorno/mese/anno.

```
1 const example = format(new Date(exampleDate), 'dd/MM/yyyy');
```

Listing 4.26: Metodo format()

### 4.3.5 mui-rte

*mui-rte* è una libreria che fornisce al contempo un «[...] editor e un visualizzatore di testo completo per MUI [...]»[24]. Nonostante sia integrabile fin da subito all’interno di un progetto, lo strumento fornisce agli sviluppatori un ampio margine di personalizzazione, ad esempio, modificando stili, strategie di autocompletamento, decoratori, barra degli strumenti e il tema.

Nell’esempio riportato dal Listing 4.27, poiché contenuto dall’elemento `Paper`, il componente `MUIRichTextEditor` “posa” su una superficie di tipo cartaceo; la sua etichetta interna è il valore dell’unica *prop* fornитagli, `label`, ovvero la stringa ‘Example...’.

```

1 <Paper variant="outlined">
2   <MUIRichTextEditor label="Example..." />

```

<sup>3</sup> </Paper>;

Listing 4.27: Componente MUIRichTextEditor



## 5. *Feature*

Infine, il quinto capitolo ha come scopo quello di offrire un resoconto dell’*output* dell’attività lavorativa personale, calando nella pratica i numerosi concetti più o meno teorici di cui si è discusso nelle precedenti sezioni. A tal fine, dopo una panoramica della procedura generale seguita ogni volta che è stato necessario realizzare un’implementazione, ovvero il “flusso di sviluppo standard”, sono state illustrate le più significative *feature* sviluppate e implementate all’interno dell’applicativo.

Va sottolineato che le implementazioni qui riportate non sono tutte quelle realizzate nel corso dei mesi; per esigenze di spazio sono state descritte solamente quelle più rappresentative dei processi svolti e delle soluzioni adottate, realizzate nell’arco di tempo che va da agosto 2022 a dicembre 2022.

La convenzione adottata per la rappresentazione ordinata delle *feature* include i seguenti elementi:

- L’iniziale della parola *feature* (“F”);
- Il numero progressivo;
- Il nome dell’implementazione.

La loro disamina avviene secondo lo schema tripartito illustrato di seguito:

1. **Requisiti:** descrizione testuale di alto livello in cui vengono presentati i requisiti specifici richiesti committente;
2. **Procedura:** descrizione, molto più precisa in termini di concetti e riferimenti, che accompagna e commenta al contempo il codice riportato;
3. **Output grafico:** *screenshot* che mostrano il *rendering* a schermo delle implementazioni effettuate.

Le *feature* dotate di un notevole livello di similarità sono state accorpate in piccole sezioni tematiche. In questi casi, la descrizione della procedura e l’illustrazione dell’*output* grafico è stata fornita solamente per la prima implementazione di quelle appartenenti a un gruppo.

## 5.1 Flusso di sviluppo standard

Attraverso l'espressione “flusso di sviluppo standard” si è deciso di rappresentare l'insieme ordinato e completo di tutti i passaggi seguiti nel corso dell'attività di sviluppo per la realizzazione di un'implementazione da un punto d'inizio zero.

Poiché tale flusso rappresenta uno scenario ideale e stereotipato, un *template*, l'*iter* realizzativo di ogni singola *feature*, differente di volta in volta in termini di punto d'inizio e di fine e di passaggi eseguiti<sup>1</sup>, può essere sempre contestualizzato all'interno di questa grande e onnicomprensiva procedura “padre”.

Lo schema riportato di seguito illustra sinteticamente la struttura del flusso di sviluppo standard.

### 1. Posizionamento componente

- (a) *Routing*
- (b) *Dashboard*
- (c) *Nesting*

### 2. Funzioni di chiamata alle API

### 3. Creazione componente

- (a) Logica
  - i. Variabili di stato
  - ii. Implementazione funzioni di chiamata alle API
  - iii. Schema
- (b) UI

### Posizionamento componente

Come riportato nella sezione 1.3, l'applicativo oggetto dell'elaborato contempla la coesistenza di più figure professionali differenti e il suo utilizzo in contemporanea da parte loro. Poiché per ogni figura professionale è previsto un flusso differente attraverso l'applicativo, quando

<sup>1</sup>Vari sono i fattori che determinano i punti d'inizio e di fine. Le richieste del committente e l'attività di sviluppo svolta in precedenza da altri membri del team costituiscono alcuni dei più influenti.

è richiesta l'introduzione di una nuova sezione, è necessario intervenire sul ***routing*** di uno specifico ruolo utente, aggiungendo il componente associato a essa.

L'introduzione di una nuova sezione richiede l'aggiornamento della ***dashboard***, il pannello di controllo che consente di navigare all'interno dell'applicativo attraverso le varie pagine.

Nel caso in cui il componente da realizzare non costituisca una sezione, ma una porzione di un più complesso e articolato componente padre, pur non avendo a che fare con *routing* e *dashboard*, è necessario posizionare la sua istanza **annidandola** all'interno dell'elemento di cui fa parte.

## Funzioni di chiamata alle API

All'interno delle sezioni che compongono l'applicativo, l'utente deve avere la possibilità di interagire con i componenti per compiere delle azioni. Come discusso nella sottosezione 4.3.1.1, ogni volta che il client si rapporta con l'interfaccia, il server viene chiamato in causa attraverso delle richieste effettuate dalle API. Per questo motivo, prima di lavorare al componente in se, è necessario provvedere alla creazione delle funzioni responsabili del dialogo della piattaforma con la macchina con cui vengono scambiati dati e informazioni.

## Creazione componente

La creazione di un componente si articola in due macro fasi. La prima prevede la realizzazione della **logica** che si cela dietro all'interfaccia grafica, ovvero di tutte quelle linee di codice che si azionano nel momento in cui l'utente interagisce con la UI (ad esempio, quando viene effettuato un click sul bottone “Apri”). Qui vengono dichiarate, rispettivamente, le variabili di stato, gli *event handler*, la variabile *schema* e, più in generale, tutte le funzioni richieste affinché il componente si comporti nel modo desiderato. La seconda macro fase consiste invece sia nella creazione dell'**interfaccia grafica** vera e propria, ossia di tutto ciò che viene rende- rizzato a schermo e con cui l'utente può interagire (ad esempio, il bottone “Apri” stesso), sia nell'implementazione della logica creata precedentemente all'interno della UI, rendendola di fatto viva.

## 5.2 F1 - F3: inserimento campo

Le *feature* F1, F2 e F3 hanno previsto l'inserimento di un campo all'interno della tabella delle informazioni di un elemento, e l'aggiornamento della modale per l'aggiunta di un nuovo elemento, e per la modifica dei dati associati a un elemento esistente.

### 5.2.1 F1: visibilità nel sito web

**Requisiti** La tabella che contiene le informazioni relative a una struttura medica deve essere dotata di un campo che riporti lo stato della visualizzazione (“Si” o “No”) sul sito web di Cerba HealthCare, nella pagina della struttura stessa.

La riga deve essere posta subito dopo il campo “Tipologia”.

Sia all'interno della modale per l'aggiunta di una nuova struttura medica che in quella di modifica dei dati di una struttura esistente, la gestione dello stato della visualizzazione sul sito web deve essere effettuata sputando un *flag*.

**Procedura** Il **primo passo** compiuto è consistito nella creazione della logica responsabile del funzionamento delle porzioni di interfaccia da inserire.

Come illustrato nel Listing 5.1, all'interno componente ModalFacility sono stati dichiarati la variabile di stato `website_visibility` e il relativo *setter* `setWebsiteVisibility()` tramite l'*hook* `useState()`.

La variabile di stato è stata inizializzata (a riga 2) con una condizione che utilizza l'operatore ternario per esprimere il seguente concetto: se è vero che la proprietà `website_visibility` dell'oggetto `facility` corrisponde<sup>2</sup> al valore booleano `true`, assegna il valore `true`, altrimenti `false`.

```
1 const [website_visibility, setWebsiteVisibility] = useState(
2   facility.website_visibility === true ? true : false
3 );
```

Listing 5.1: Variabile di stato `website_visibility`

---

<sup>2</sup>Più precisamente, l'operatore *strict equality* (`==`) verifica che i due dati fra i quali è posto siano uguali sia per valore e che per tipo.

Inoltre, è stato aggiornato il contenuto delle variabili `schema` e `form`<sup>3</sup> con l'aggiunta, in entrambi i casi, di una linea di codice relativa alla visibilità nel sito web.

Poiché la variabile `website_visibility` opera con valori booleani, all'interno dell'oggetto `schema`, visibile nel Listing 5.2, l'espressione `website_visibility: yup.boolean()` (a riga 3) verifica che accada proprio questo.

```

1 const schema = yup.object().shape({
2   // [...]
3   website_visibility: yup.boolean(),
4   // [...]
5 });

```

Listing 5.2: Aggiornamento variabile `schema`

L'oggetto `form`, visibile nel Listing 5.3, fa invece parte dell'*handler* `handleSave()`<sup>4</sup>, funzione che ha come compito il salvataggio dei dati una volta terminato il loro inserimento, lato client, nella modale. Tale oggetto vien passato come argomento alla funzione `validate()` per una validazione “generale” di quanto inserito nella finestra di dialogo; se i dati sono corretti viene eseguito il codice di `handleSave()` effettivamente responsabile del salvataggio dei dati<sup>5</sup>. L'aggiunta di `website_visibility` (a riga 4) fra le proprietà dell'oggetto `form` va a segnalare che anche quest'ultima fa ora parte di quelle a lui appartenti.

```

1 const handleSave = async () => {
2   const form = {
3     // [...]
4     website_visibility,
5     // [...]
6   };
7   // [...];
8 };

```

Listing 5.3: Aggiornamento variabile `form`

---

<sup>3</sup>Generalmente, assieme a `schema` e `form` viene aggiornato anche l'oggetto `body`, ciò che viene passato all'API innescata al click del bottone “Salva” della modale. In questo caso, ciò non è avvenuto a causa della struttura della variabile `body` stessa, che non ha necessitato dell'aggiunta della proprietà `website_visibility`.

<sup>4</sup>Attraverso le `props`, la funzione `handleSave()` viene passata componente `ModalFacility` per essere utilizzata ogni volta che si clicca sul tasto salva.

<sup>5</sup>Il codice dell'*handler* `handleSave()` che si occupa dell'effettivo salvataggio dei dati, qui non riportato, si avvale della sintassi `try...catch`. Per un esempio, si veda la sottosezione 5.5.1.

Il **secondo passo**, illustrato nel Listing 5.4, ha invece previsto due operazioni. Da un lato, è stata creata la porzione di UI della modale atta alla gestione, lato client, dello stato relativo alla visualizzazione della struttura medica nel sito web. Il componente padre Grid<sup>6</sup> (da riga 3 a 21) contiene al suo interno altri due componenti figli Grid, ognuno dei quali dotato, a sua volta, di un componente figlio. Rispettivamente, il primo Grid racchiude Typography, componente usato per la scrittura del nome della cella (“Visibilità”, a riga 5), mentre il secondo FormGroup, FormControlLabel e Checkbox, i componenti responsabili della struttura della checkbox (da riga 9 a 20).

Dall’altro lato, la porzione di UI appena descritta è stata dotata della logica discussa inizialmente. Le *props* fornite al componente Checkbox sono *props*:

- checked (a riga 13): il suo valore è la variabile di stato website\_visibility;
- onChange (a riga 14): il suo valore è una funzione che, sfruttando il funzionamento del setter setWebsiteVisibility(), ha il compito di sostituire il precedente valore della variabile di stato website\_visibility.

Queste linee di codice concludono le modifiche alla modale per l’aggiunta di una nuova struttura medica e per la modifica dei dati di una struttura esistente visualizzati nella relativa tabella delle informazioni.

```

1 <Dialog>
2   {/* [...] */}
3   <Grid>
4     <Grid item xs={12} sx={{ mb: -2 }}>
5       <Typography variant="subtitle2">Visibilità</Typography>
6     </Grid>
7
8     <Grid item xs={12}>
9       <FormGroup>
10      <FormControlLabel
11        control={
12          <Checkbox
13            checked={website_visibility}
14            onChange={e => setWebsiteVisibility(e.target.checked)}>

```

---

<sup>6</sup>Il componente padre Grid e tutti i suoi figli (Grid, Typography, FormGroup, FormControlLabel, <Checkbox>) sono a loro volta figli del componente Dialog.

```

15      />
16    }
17    label="Struttura visibile su sito web"
18  />
19  </FormGroup>
20  </Grid>
21  </Grid>
22 /* [...] */
23 </Dialog>;

```

Listing 5.4: *Checkbox* modifica visibilità sito web

Il **terzo e ultimo passo** compiuto, quello che segna la chiusura del processo nella sua interezza, ha riguardato l’aggiornamento della UI della tabella delle informazioni (della struttura medica, fra cui anche lo stato della visualizzazione sul sito web), alla quale è stata poi fornita la logica per il funzionamento. Tutto ciò è stato svolto all’interno del componente TabInfo ed è illustrato nel Listing 5.5.

Il componente TableRow (da riga 3 a 6) rappresenta sia la nuova riga aggiunta alla tabella già esistente<sup>7</sup>, che un elemento padre. I suoi due componenti figli altro non sono che le celle della riga in cui saranno presentate le informazioni:

- TableCell (a riga 4): contiene il nome della riga appena aggiunta, “Visibilità sul sito web”;
- TableCell (a riga 5): contiene una condizione che utilizza l’operatore ternario per esprimere il seguente concetto: se è vero che il valore della proprietà website\_visibility dell’oggetto facility è true assegna come la valore la stringa "Si ", altrimenti la stringa "No".

```

1 <Table>
2 /* [...] */
3 <TableRow>
4   <TableCell>Struttura visibile su sito web</TableCell>
5   <TableCell>{facility.website_visibility ? 'Si' : 'No'}</TableCell>
6 </TableRow>
7 /* [...] */

```

---

<sup>7</sup>Il componente TableRow e i suoi figli (TableCell) sono a loro volta figli del componente Table.

8 </Table>;

Listing 5.5: Riga visibilità sito web

## 5.2.2 F2: e-mail

**Requisiti** La tabella che contiene le informazioni relative alla singola struttura medica deve essere dotata di uno nuovo campo che riporti la sua e-mail.

La riga deve essere posta fra i campi “Brand storico” e “Telefono”.

All’interno della modale di modifica dei dati di una struttura esistente, il campo di testo dell’e-mail deve presentare l’e-mail qualora essa sia presente ed essere vuoto in caso contrario; invece, all’interno della modale di aggiunta di una nuova struttura, il campo di testo dell’e-mail deve semplicemente essere vuoto.

## 5.2.3 F3: link referti e portale prenotazione

**Requisiti** La tabella che contiene le informazioni relative alla singola struttura medica deve essere dotata di uno nuovo campo che riporti il link per l’inserimento dei referti.

La riga deve essere posta in coda ai campi esistenti.

All’interno della modale, il campo di testo del link dei referti deve presentare il link, qualora esso sia presente e si sia in fase di modifica dei dati di una struttura esistente, o essere vuoto, qualora esso sia assente e si sia in fase di aggiunta di una nuova struttura.

La tabella che contiene le informazioni relative alla singola *business unit* (BU) deve essere dotata di due nuovi campi, uno che riporti il link del portale di prenotazione, e uno che riporti il link del portale di prenotazione SSN.

All’interno della modale, il campo di testo di entrambi i link deve presentare il link, qualora esso sia presente e si sia in fase di modifica dei dati di una struttura esistente, o essere vuoto, qualora esso sia assente e si sia in fase di aggiunta di una nuova struttura.

## Output grafico

Figura 5.1: Modale aggiunta nuova struttura

Figura 5.2: Modale modifica struttura esistente

Figura 5.3: Tabella informazioni struttura

Figura 5.4: Modale aggiunta nuova BU

Figura 5.5: Modale modifica BU esistente

The screenshot shows the Cerba HealthCare ITALIA software interface. At the top, there is a navigation bar with tabs: INFORMAZIONI, PRESIDIO, ORGANIZZAZIONE INTERNA, BUSINESS UNIT (which is currently selected), PARTNER, LABORATORIO, and CORRIERI. On the far right of the top bar are buttons for ADMIN STRUTTURE and a user icon.

The main content area is titled "Strutture" and shows a list of Business Units (BU). A sidebar on the left lists various structure types: Utenti, Partner, Ragni sociali, Conti bancari, Corrieri, Brand storici, Professionisti, Scadenzario, and Agenzie. Below this is a "Sito Web" section with a dropdown menu.

A specific Business Unit, "Medicina dello Sport", is highlighted. Its details are shown in a modal window:

- Business unit:** Medicina dello Sport
- State:** Chiuso
- Link portale prenotazione:** <https://www.youtube.com/watch?v=1hrRSUfaADM>
- Autorizzazione sanitaria:** Accreditato
- Ore:** Non ci sono orari

At the bottom of the modal window are buttons for MODIFICA and ELIMINA.

In the bottom right corner of the main interface, there is a small DigitalFactory logo with the text "v0.56.0 DEVELOPMENT VERSION".

Figura 5.6: Tabella BU esistenti e tabella informazioni BU

## 5.3 F4: modifica tabella contratti di locazione

**Requisiti** La tabella dei contratti di locazione deve essere modificata in modo da poterne visualizzare e gestire di multipli.

Deve essere possibile aggiungere e archiviare contratti.

La tabella deve mostrare i contratti esistenti in una lista semplice e potersi espandere per mostrare i dettagli di ognuno.

**Procedura** Il **primo passo** compiuto è consistito nell'esecuzione delle modifiche riguardanti il funzionamento delle modali. Poiché il codice preesistente permetteva già la modifica dei contratti di locazione esistenti, le introduzioni effettuate hanno riguardato la possibilità di aggiungerne di nuovi e la loro archiviazione.

*In primis*, ciò ha necessitato la creazione delle funzioni `createRentalAgreement()` e `deleteRentalAgreement()`, le funzioni che effettuano le chiamate alle API visibili nel Listing 5.6. All'interno del loro corpo (alle righe 2 e 5) sono state utilizzate, rispettivamente, le funzioni `put()` e `del()`, funzioni che hanno il compito di effettuare richieste avvalendosi dei metodi PUT e DELETE.

```

1 export const createRentalAgreement = (uuid, body) =>
2   post('/facility/${uuid}/rental-agreement', body, { baseUrl: 'facilities' });
3
4 export const deleteRentalAgreement = (uuid, body) =>
5   del('/rental-agreement/${uuid}', body, { baseUrl: 'facilities' });

```

Listing 5.6: Funzioni di chiamata API contratti

In secondo luogo, la gestione delle modali ha richiesto, a seconda dei casi, la creazione o l'aggiornamento degli *handler*, le funzioni innescate al click di uno specifico bottone<sup>8</sup>; questi possono essere divisi in due gruppi a seconda dell'obiettivo che si pongono.

Il primo ha come scopo l'apertura della modale ed è formato da

`handleCreateRentalAgreement()` e `handleDeleteRentalAgreement()`, gli *handler* mostrati nel Listing 5.7, dichiarati all'interno del componente `RentalAgreement`. Il loro funzionamento è sostanzialmente identico e le operazioni che eseguono sono:

<sup>8</sup>Tendenzialmente, nel corpo degli *handler* sono invocate le funzioni che effettuano le chiamate alle API.

1. L'impostazione dei dati della modale, attraverso l'uso del *setter* `setData()` (alle righe 2 e 9);
2. L'apertura della finestra di dialogo, con l'utilizzo della funzione `open()` (alle righe 3 e 10).

Le differenze riguardano la modale con cui operano e i dati sui quali essa viene impostata. Infatti, se la funzione `handleCreateRentalAgreement()` ha a che fare con la finestra di dialogo `modalRentalAgreement`<sup>9</sup> e ne imposta i dati su un oggetto vuoto, `handleDeleteRentalAgreement()` lavora invece con `modalConfirmDeleteRentalAgreement` e ne imposta i dati sull'oggetto `rentalAgreement`<sup>10</sup>. L'oggetto di quest'ultima finestra di dialogo è stato definito attraverso l'uso dell'*hook* `useModal()` (a riga 6).

```

1 const handleCreateRentalAgreement = () => {
2   modalRentalAgreement.setData({ rentalAgreement: {} });
3   modalRentalAgreement.open();
4 };
5
6 const modalConfirmDeleteRentalAgreement = useModal();
7
8 const handleDeleteRentalAgreement = rentalAgreement => {
9   modalConfirmDeleteRentalAgreement.setData({ rentalAgreement });
10  modalConfirmDeleteRentalAgreement.open();
11 };

```

Listing 5.7: *Handler* apertura modale

Il secondo gruppo di *handler* ha invece come obiettivo l'effettiva manipolazione dei dati ed è formato da `handleSaveAgreement()` e `handleArchiveConfirmRentalAgreement()`.

Entrambe le funzioni adempiono al loro compito invocando, all'interno del loro corpo, una delle funzioni di chiamata alle API definite poco fa.

Da un lato, come mostrato nel Listing 5.8, l'*handler* `handleSaveAgreement()` (già esistente all'interno del componente `ModalRentalAgreement` in passato) è stato aggiornato con l'in-

---

<sup>9</sup>La definizione della variabile della modale `modalRentalAgreement` non è riportata nel Listing 5.7 perché già parte del codice preesistente.

<sup>10</sup>La funzione `handleDeleteRentalAgreement()` si aspetta il parametro `rentalAgreement` in *input*. Per questo motivo, il valore di `rentalAgreement` corrisponde all'argomento fornito durante l'invocazione della funzione `handleDeleteRentalAgreement()`.

troduzione della possibilità di aggiungere un nuovo contratto di locazione. La già esistente riga di codice atta alla gestione della modifica dei contratti, infatti, è stata resa parte di una condizione che si avvale della sintassi `if...else` (da riga 3 a 7) per esprimere il seguente concetto: se è presente la proprietà `uuid` dell'oggetto `rentalAgreement` esegui la funzione `editRentalAgreement()` passandole come parametri la proprietà `uuid` stessa e la variabile `body`; in caso contrario esegui la funzione `createRentalAgreement()` passandole come parametri le variabili `uuidFacility` e `body`.

In altre parole, se esiste un contratto di locazione eseguine la modifica, altrimenti creane uno nuovo.

```

1 const handleSaveAgreement = async () => {
2   // [...]
3   if (rentalAgreement.uuid) {
4     await editRentalAgreement(rentalAgreement.uuid, body);
5   } else {
6     await createRentalAgreement(uuidFacility, body);
7   }
8   // [...]
9 };

```

Listing 5.8: *Handler* aggiunta nuovo contratto

Dall'altro lato, come illustrato nel Listing 5.9, è stato creato, di nuovo all'interno del componente `RentalAgreement`, l'*handler* `handleArchiveConfirmRentalAgreement()` per l'archiviazione dei contratti di locazione. Questo esegue l'archiviazione del contratto, grazie alla funzione `deleteRentalAgreement()`, e l'aggiornamento della lista dei contratti di locazione, tramite `fetchRentalAgreements()`.

```

1 const handleArchiveConfirmRentalAgreement = async () => {
2   await deleteRentalAgreement(
3     modalConfirmDeleteRentalAgreement.data.rentalAgreement.uuid
4   );
5   fetchRentalAgreements();
6 };

```

Listing 5.9: *Handler* archiviazione contratto

Il **secondo passo** compiuto, invece, è consistito nell'aggiornamento della UI del compo-

nente `RentalAgreement`, elemento responsabile dell'intera tabella dei contratti di locazione.

Come possibile vedere nel Listing 5.10, un componente `Card` (da riga 2 a 28) è stato utilizzato per racchiudere tutti i componenti figli che formano la struttura della tabella.

`CardHeader`, il primo di essi, ha come compito il *rendering* dell'*header* e le sue *props* sono:

- `subheader` (a riga 4): il suo valore è la stringa "Contratti locazione";
- `action` (da riga 5 a 12): il suo valore è il componente `Button`, il quale racchiude il testo "Aggiungi" ed è dotato, a sua volta, delle *props*:
  - `onClick`: il suo valore è l'*handler* `handleCreateRentalAgreement`;
  - `startIcon`: il suo valore è il componente `AddIcon`.

Il secondo componente figlio di `Card` è `Divider` (a riga 15), elemento che stabilisce un confine fra l'*header* e tutto ciò che sta al di sotto di lui.

La struttura di `Card` termina con il componente `Table`, terzo ed ultimo elemento che si occupa del *rendering* del corpo della tabella (da riga 18 a 29). Questo racchiude un componente `TableBody` all'interno del quale viene effettuato un *mapping*: l'iteratore `map()` è utilizzato sull'`array rentalAgreements` affinché tutti gli oggetti che lo compongono vengano visualizzati grazie all'istanza del componente `TabBuildingRentalAgreementTable`. Le *props* passate a quest'ultima sono:

- `key`: il suo valore è la proprietà `uuid` dell'oggetto `el`;
- `el`: il singolo contratto che, di volta in volta, viene mappato sul componente. Il suo valore è l'oggetto `el`;
- `handleEditRentalAgreement`: il suo valore è l'*handler* `handleEditRentalAgreement()`;
- `handleDeleteRentalAgreement`: il suo valore è l'*handler* `handleDeleteRentalAgreement()`.

Le ultime due *props* forniscono le funzioni invocate dai bottoni contenuti nel componente.

Inoltre, è stato necessario introdurre il componente `ModalConfirm`, elemento responsabile della restituzione dell'ulteriore finestra di dialogo che compare per chiedere all'utente la conferma prima di eseguire l'archiviazione di un contratto.

Tramite una condizione che si avvale dell'operatore logico AND (da riga 32 a 39) viene stabilito che, se il valore della proprietà `isOpen` dell'oggetto

`modalConfirmDeleteRentalAgreement` è vero, allora viene eseguito il *rendering* del componente. In tal caso, le *props* fornite a `ModalConfirm` sono:

- `modal`: il suo valore è l'oggetto `modalConfirmDeleteRentalAgreement`;
- `title`: il suo valore è la stringa '`Archivia contratto locazione`';
- `text`: il suo valore è un'espressione che stabilisce la composizione di una stringa;
- `handleConfirm`: il suo valore è l'*handler* `handleArchiveConfirmRentalAgreement()`.

La funzione passata alla *prop* `handleConfirm` è quella che viene invocata dal bottone del componente.

```

1  <>
2    <Card>
3      <CardHeader
4        subheader="Contratti locazione"
5        action={
6          <Button
7            size="small" startIcon={<AddIcon />}
8            onClick={handleCreateRentalAgreement}
9          >
10         Aggiungi
11       </Button>
12     }
13   ></CardHeader>
14
15   <Divider />
16
17   <Table size="small">
18     <TableBody>
19       {rentalAgreements.map(el => (
20         <TabBuildingRentalAgreementTable
21           key={el.uuid} el={el}
22           handleEditRentalAgreement={handleEditRentalAgreement}
23           handleDeleteRentalAgreement={handleDeleteRentalAgreement}
24         />
25       )));
26     </TableBody>

```

```

27   </Table>
28   </Card>
29
30   {/* [...] */}
31
32   {modalConfirmDeleteRentalAgreement.isOpen && (
33     <ModalConfirm
34       modal={modalConfirmDeleteRentalAgreement}
35       title="Archivia contratto locazione"
36       text={'Archiviare il contratto locazione: ${'
37         modalConfirmDeleteRentalAgreement.data.rentalAgreement.type.label?'}
38       handleConfirm={handleArchiveConfirmRentalAgreement}
39     />
40   )}
41 </>;

```

Listing 5.10: Tabella contratti esistenti

Il **terzo passo** è consistito nella gestione del corpo della tabella per la visualizzazione delle informazioni dei contratti di locazione. Ciò è stato fatto attraverso l'implementazione di un nuovo componente creato *ad hoc*, TabBuildingRentalAgreementTable, del quale era stata utilizzata precedentemente l'istanza all'interno del *mapping*.

Come prassi vuole, la definizione di tale elemento ha previsto l'importazione delle *props* passate alla sua istanza; il Listing 5.11 mostra che ciò è stato fatto, attraverso la tecnica della destrutturazione, per el, handleEditRentalAgreement() e handleDeleteRentalAgreement() (a riga 2).

```

1 const TabBuildingRentalAgreementTable = ({
2   el, handleEditRentalAgreement, handleDeleteRentalAgreement,
3 }) => {
4   // [...]
5   return(
6     // [...]
7   )
8 };

```

Listing 5.11: Componente TabBuildingRentalAgreementTable

Anche in questo caso, delle modifiche a livello di logica hanno costituito lo *step* inizia-

le. Poiché fra i requisiti era presente l'espandibilità di ogni riga della tabella, la definizione del componente è stata seguita dalla gestione di questa dinamica<sup>11</sup>, che viene illustrata nel Listing 5.12.

Il concetto di espansione è stato implementato grazie alla variabile di stato `toggleOpen` e al relativo *setter* `setToggleOpen()`, dichiarati attraverso l'*hook* `useState()` (a riga 1). Poiché, salvo click da parte dell'utente, le righe della tabella appaiono nella pagina come non espanso, `toggleOpen` è stata inizializzata con il valore booleano `false`.

Inoltre, è possibile vedere l'*handler* `handleOpenRentalAgreement`, la funzione responsabile del funzionamento del meccanismo di espansione. Questa sfrutta il *setter* `setToggleOpen()` per sostituire il precedente valore della variabile di stato `toggleOpen` con il suo opposto ogni volta che viene invocata.

```

1  const [toggleOpen, setToggleOpen] = useState(false);
2
3  const handleOpenRentalAgreement = () => {
4      setToggleOpen(!toggleOpen);
5  };

```

Listing 5.12: Dinamica apertura riga tabella contratti

Il **quarto** e **ultimo** passo è consistito nella creazione della UI del neonato componente `TabBuildingRentalAgreementTable`.

Il Listing 5.13 mostra come il tag padre `<>` sia stato utilizzato come contenitore per due grandi componenti figli `TableRow`. Questi costituiscono le due righe che, di volta in volta, racchiudono le informazioni dei contratti di locazione esistenti. Se la prima deve permettere all'utente di visualizzare, anche senza l'espansione della riga, solo alcuni dati del contratto e i bottoni per la sua gestione, la seconda deve invece mostrare, una volta espansa la riga, tutte le informazioni del contratto nel dettaglio.

Entrambi i componenti `TableRow` contengono, a loro volta, molti elementi figli; all'interno del primo (da riga 2 a 41) sono presenti quattro componenti `TableCell`:

- `TableCell` (da riga 3 a 7): contiene il componente `Typography` che effettua il *rendering* di un sottotitolo. Il sottotitolo ritornato è quello risultante dalla condizione che si avvale dell'operatore ternario (a riga 5); questa esprime il seguente concetto: se la pro-

---

<sup>11</sup>La gestione della dinamica di espansione è stata effettuata all'interno del corpo del componente `TabBuildingRentalAgreementTable`.

prietà `label` dell’oggetto `type` (contenuto dall’oggetto `e1`) è vera, allora restituisci tale proprietà, altrimenti una stringa dal valore ‘-’;

- `TableCell` (a riga 9): contiene una condizione che si avvale dell’operatore logico AND per esprimere il seguente concetto: solo se la proprietà `archived` dell’oggetto `e1` è vera, restituisci la stringa ‘(archiviato)’;
- `TableCell` (da riga 11 a 25): contiene il componente `Button`, il bottone necessario all’espansione della riga. Le sue *props* più importanti sono:
  - `onClick` (a riga 14): il suo valore è `handleOpenRentalAgreement()`;
  - `startIcon` (a riga 15): il suo valore è una condizione che si avvale dell’operatore ternario per esprimere il seguente concetto: se la variabile di stato `toggleOpen` ha come valore il `true`, restituisci il componente `KeyBoardArrowUpIcon`, altrimenti `KeyBoardArrowDownIcon`. In altre parole, se la riga è espansa restituisci una freccia che punta verso l’alto, altrimenti, una freccia che punta verso il basso.

All’interno del componente viene effettuato il *rendering* del testo “Mostra”;

- `TableCell` (da riga 27 a 40): contiene due componenti `IconButton`, ovvero i bottoni atti alla manipolazione dei contratti di locazione. Entrambi sono dotati della *prop* `onClick`, il cui valore è però diverso a seconda della casistica: quello del bottone che si occupa della modifica è l’*handler* `handleEditRentalAgreement()`, mentre quello del bottone attivo all’archiviazione è `handleDeleteRentalAgreement()`. A entrambe le funzioni viene passato come argomento l’oggetto `e1`, il contratto che di volta in volta sarà modificato o archiviato.

Inoltre, il bottone per la modifica contiene il componente `EditIcon`, mentre quello per l’archiviazione `DeleteIcon`.

All’interno del secondo componente `TableRow` (da riga 43 a 60) è invece presente un solo `TableCell` che contiene a sua volta il componente `Collapse` (da riga 45 a 58), elemento il cui compito è l’effettiva gestione della dinamica di apertura e chiusura della riga della tabella; ciò avviene grazie alla *prop* `in`, il cui valore è la variabile di stato `toggleOpen`.

Le informazioni dettagliate di ogni contratto di locazione vengono mostrate attraverso un’ulteriore tabella, visibile solamente a riga espansa, contenuta dal componente `TableBody`<sup>12</sup>.

---

<sup>12</sup>La struttura della tabella, incomincia in realtà con il componente `TableContainer` (da riga 46 a 57).

Questo è composto da tanti componenti TableRow quante sono le proprietà dell'oggetto el, ovvero, quante sono le informazioni del contratto che si desidera vengano mostrate all'interno della tabella. Ogni riga è formata a sua volta da due componenti TableCell, una cella per la visualizzazione del tipo di informazione riportata, e una per il dato associato al tipo di informazione.

```
1  <>
2    <TableRow>
3      <TableCell>
4        <Typography variant="subtitle2">
5          {el.type.label ? el.type.label : '—'}
6        </Typography>
7      </TableCell>
8
9      <TableCell align="right">{el.archived && '(archiviato)'}</TableCell>
10
11     <TableCell>
12       <Button
13         sx={{ ml: 8, mr: 2 }} size="small"
14         onClick={handleOpenRentalAgreement}
15         startIcon={
16           toggleOpen ? (
17             <KeyboardArrowUpIcon fontSize="inherit" />
18           ) : (
19             <KeyboardArrowDownIcon fontSize="inherit" />
20           )
21         }
22       >
23         Mostra
24       </Button>
25     </TableCell>
26
27     <TableCell>
28       <IconButton
29         size="small"
30         onClick={() => handleEditRentalAgreement(el)}
31       >
32         <EditIcon fontSize="inherit" />
```

```
33     </ IconButton>
34
35     < IconButton
36         size="small" onClick={() => handleDeleteRentalAgreement(el)}
37     >
38         < ArchiveIcon fontSize="inherit" />
39     </ IconButton>
40   </ TableCell>
41 </ TableRow>
42
43 < TableRow>
44   < TableCell sx={{ p: toggleOpen ? 1 : 0 }} colSpan={4}>
45     < Collapse in={toggleOpen} timeout="auto" unmountOnExit>
46       < TableContainer component={Paper}>
47         < Table size="small">
48           < TableBody >
49             {/* [...] */}
50             < TableRow >
51               < TableCell > Note </ TableCell >
52               < TableCell > {el.renewal_note || '-' } </ TableCell >
53             < / TableRow >
54             {/* [...] */}
55           < / TableBody >
56         < / Table >
57       < / TableContainer >
58     < / Collapse >
59   < / TableCell >
60 < / TableRow >
61 < />;
```

Listing 5.13: Riga tabella contratti

### Output grafico

Figura 5.7: Modale aggiunta nuovo contratto

Figura 5.8: Modale archiviazione contratto esistente

Figura 5.9: Tabella contratti non espansa

Figura 5.10: Tabella contratti con riga espansa

## 5.4 F5: inserimento campo ricerca

La *feature* F5 è stata caratterizzata da un certo livello di ridondanza: dopo la scrittura della nuova porzione di codice per la prima volta, l'implementazione è consistita semplicemente nella sua riproduzione dove previsto.

Nonostante piccole modifiche siano state operate di volta in volta ove necessario, i cambiamenti effettuati non rappresentano un qualcosa di così rilevante da giustificare l'inserimento dell'*output* grafico di ogni situazione in cui l'implementazione è stata effettuata.

**Requisiti** L'*header* della tabella di ognuna delle schede che formano la sezione “Ruoli Utente” (le schede sono: *REO*, *Area Manager*, *Center Manager*, *Data Manager*, *Coordinatore Turni* e *Coordinatore Compensi*) deve essere dotato di un campo di ricerca. Questo deve essere in grado di filtrare gli utenti per nome e/o cognome e posizionato alla sinistra del bottone per l'aggiunta di un ruolo utente.

**Procedura** L'intero processo che ha portato all'implementazione del campo di ricerca all'interno della scheda del ruolo utente REO è avvenuto all'interno dello stesso componente, TabReo.

Il **primo step** è consistito nella creazione della logica responsabile del suo funzionamento. Ciò ha necessitato, *in primis*, la dichiarazione della variabile di stato *filter* e del relativo *setter* *setFilter()* attraverso l'*hook* *useState()*. Tale variabile ha il compito di “memorizzare” il testo immesso, lato client, all'interno della casella di testo del campo di ricerca. Poiché, salvo introduzione di testo da parte dell'utente, il campo di ricerca si presenta come vuoto, la variabile *filter* è inizializzata con una stringa vuota ”. Il contenuto di tale variabile viene aggiornato in continuazione con quanto scritto lato client attraverso il *setter* *setFilter()*.

```
1 const [filter, setFilter] = useState('');
```

Listing 5.14: Variabile di stato *filter*

In secondo luogo, è stata creata la variabile più propriamente responsabile del funzionamento della dinamica di ricerca, *filtered*. Questa implementa l'*hook* *useMemo()* per il miglioramento delle *performance* di esecuzione della ricerca.

All'interno del corpo di questa funzione è stata dichiarata la variabile *\_filter* (a riga 2): l'utilizzo dei metodi *toLowerCase()* e *trim()* sulla variabile di stato *filter*, assicura che il testo

(in essa inserito ed immagazzionato) venga archiviato all'interno di un'altra variabile, `_filter`, trasformato in minuscolo.

In seguito è stata dichiarata la variabile `filtered`: grazie allo *spread operator* viene effettuata una copia di tutti i ruoli utente immagazzinati nella variabile `reos`<sup>13</sup>.

La condizione composta da un solo `if` (da riga 6 a 12) costituisce la sezione più importante della variabile `filtered`; questa esprime il seguente concetto: se la lunghezza di `_filter` è maggiore di 0, sovrascrivi il precedente valore di `filtered` con i soli utenti frutto del filtraggio, eseguito tramite l'iteratore `filter()`, sull'oggetto `filtered`.

Tale filtraggio, attraverso i metodi `toLowerCase()` e `includes()` sull'espressione che li precede, seleziona solo gli elementi (contenuti dalla variabile `filtered`) il cui nome e cognome o il cognome e nome in minuscolo include quando contenuto in `_filter`.

In altre parole, vengono restituiti gli utenti del ruolo utente REO che corrispondono a quanto scritto all'interno del campo di ricerca.

```

1 const filtered = useMemo(() => {
2   const _filter = filter.toLowerCase().trim();
3
4   let filtered = [...reos];
5
6   if (_filter.length > 0) {
7     filtered = filtered.filter(
8       el =>
9         `${el.name} ${el.surname}`.toLowerCase().includes(_filter) ||
10        `${el.surname} ${el.name}`.toLowerCase().includes(_filter)
11      );
12    }
13
14   return filtered;
15 }, [reos, filter]);

```

Listing 5.15: Variabile `filtered`

**Il secondo passaggio** è consistito nella creazione della porzione di interfaccia del campo di ricerca.

---

<sup>13</sup>La variabile `reos`, assente nel Listing 5.16 perché già presente all'interno del codice, è frutto della chiamata all'API `getReos()`.

L’ oggetto `table`<sup>14</sup> è stata modificata aggiungendo, all’interno della proprietà `action`, il componente `TextField`. Fra le molteplici *props* di cui è dotato le più importanti sono:

- `label`: il suo valore è la stringa "Filtrà per nome o cognome";
- `placeholder`: il suo valore è la stringa "Cerca";
- `value`: il suo valore è la variabile di stato `filter`;
- `onChange`: il suo valore è una funzione che permette di sostituire il contenuto della variabile di stato `filter` sfruttando il *setter* `setFilter()`. Il nuovo contenuto sarà di volta in volta la proprietà `value` dell’oggetto `target` ( contenuto a sua volta dall’oggetto `e`);
- `InputProps`: il suo valore è l’oggetto `endAdornment`, il quale contiene al suo interno i componenti `InputAdornment`, `IconButton` e `ClearButton`. Il componente `IconButton` è stato dotato della *prop* `onClick` associata, di nuovo, al *setter* `setFilter()`. In questo caso, però, il suo argomento è una stringa vuota "" che serve per “azzerare” il valore della variabile `filter` ogni volta che viene effettuato click su quel bottone.

Infine, è stato modificato l’*array* protagonista del *mapping* utilizzato per popolare la proprietà `rows`, le righe, dell’oggetto `table`, della tabella. La sostituzione del precedente con `filtered` fa in modo che, ogni qualvolta viene effettuata una ricerca scrivendo all’interno dell’apposito campo, vengano restituite, nella tabella dei REO, solamente le righe delle che corrispondono ai criteri specificati per il filtraggio.

```

1 const table = {
2   title: 'REO',
3   action: (
4     <>
5       <TextField
6         sx={{ width: 300, mr: 2 }} size="small" variant="filled"
7         label="Filtrà per nome o cognome"
8         placeholder="Cerca..."
9         value={filter}
10        onChange={e => setFilter(e.target.value)}
11        InputProps={{

```

---

<sup>14</sup>Il *rendering* dell’ oggetto `table` avviene grazie al componente `SuperTable` (all’interno del metodo `render()`). Per una spiegazione più approfondita, si veda la sottosezione 5.5.1.

```
12   endAdornment: (
13     <InputAdornment
14       position="end"
15       style={{ display: filter.length > 0 ? 'flex' : 'none' }}
16     >
17       <IconButton onClick={() => setFilter('')}>
18         <ClearIcon />
19       </IconButton>
20     </InputAdornment>
21   ) ,
22 }
23 />
24 /* [...] */
25 </>
26 ) ,
27 // [...]
28 rows: filtered.map(
29   // [...]
30 ) ,
31 };
```

Listing 5.16: Aggiornamento della variabile `table`

## Output grafico

Name	Email	Telefono
Tagliarino Martina Tagliarino	martina.tagliarino@gmail.com	+393481632744
Reo Facilities	facilities.reo@link-up.it	+39123456789
Pasinetti Umberto	pasinetti96@gmail.com	+3458484089

Figura 5.11: Tabella REO campo ricerca vuoto

Name	Email	Telefono
Pasinetti Umberto	pasinetti96@gmail.com	+3458484089

Figura 5.12: Tabella REO campo ricerca riempito

## 5.5 F6 - F10: inserimento sezione

Le *feature* F6, F7, F8, F9 e F10 hanno previsto l'inserimento di una nuova sezione.

Da un lato, le sezioni di F6 e F7 sono state formate da una pagina per la visualizzazione dell'elenco degli elementi esistenti, e dalle modali per l'aggiunta, la modifica e l'eliminazione di un elemento. Nonostante dei cambiamenti siano stati effettuati ove necessario, le prime due *feature* sono molto simili fra loro; per questo motivo si è optato di riportare solamente l'*output* grafico relativo all'implementazione di F6.

Dall'altro, le sezioni di F8, F9 e F10 sono state composte da una pagina per la visualizzazione dell'elenco degli elementi, e da una pagina di dettaglio dedicata a ognuno di essi. A seconda dei casi, tramite le modali è possibile aggiungere, modificare o eliminare dati dell'elemento o l'elemento stesso. Anche qui le varie *feature* non differiscono di molto, ma rispetto a quanto fatto per il precedente "gruppo" di implementazioni è stato riportato sia l'*output* grafico di F8 che di F10.

### 5.5.1 F6: "Magazine"

**Requisiti** Aggiungere la sezione "Magazine" fra quelle previste per il ruolo utente *marketing admin*.

Al suo interno, una tabella deve mostrare i magazine esistenti in una lista semplice e riportarne le proprietà "Titolo", "Descrizione", "Data", "Immagine" e "Documento".

Deve essere possibile aggiungere o eliminare un magazine.

Deve essere possibile effettuare il download del documento.

Solamente per la proprietà "Descrizione" deve essere fornita la possibilità di formattazione.

L'*header* della tabella dei magazine deve essere dotato di un campo di ricerca. Questo deve essere in grado di filtrare per titolo e posizionato alla sinistra del bottone per l'aggiunta di un nuovo magazine.

**Procedura** Lo **step iniziale** ha riguardato l'aggiornamento del *routing*, operazione che ha comportato l'esecuzione di alcune modifiche all'interno del componente responsabile della navigazione fra le pagine visualizzate dal ruolo utente *marketing admin*, ovvero `MarketingAdminRoutes`. Come illustrato dal Listing 5.17, il suo compito è sia il *rendering* della *dashboard*, ovvero del componente `DashboardMarketingAdmin`, che il contenimento dei vari componenti che definis-

scono il percorso che l’utente può seguire muovendosi nella piattaforma. La modifica al *routing* ha necessitato l’inserimento, all’interno del componente padre *Switch* (da riga 3 a 9), di un nuovo componente figlio *Route*, il quale racchiude a sua volta *PageMagazines*, l’istanza del nuovo componente che verrà restituito cliccando sul link della sezione “Magazines”. L’istanza di *PageMagazines* è dotata della *prop path*, il cui valore è la stringa “/magazine”.

In altre parole, la navigazione può ora avvenire, oltre che fra le vecchie pagine (o meglio, fra i vecchi componenti), anche su *PageMagazines*.

```

1 <DashboardMarketingAdmin>
2   <Suspense fallback={<Loading />}>
3     <Switch>
4       {/* [...] */}
5       <Route path="/magazine">
6         <PageMagazines />
7       </Route>
8       {/* [...] */}
9     </Switch>
10   </Suspense>
11 </DashboardMarketingAdmin>;

```

Listing 5.17: *Routing* di *PageMagazines*

**Il secondo passo** è consistito nell’aggiornamento della *dashboard*, il “pannello” di controllo del ruolo utente *marketing admin*; ciò ha richiesto dei ritocchi al componente *DashBoardMarketingAdmin*. Per prima cosa, come possibile vedere nel Listing 5.18, l’oggetto che contiene i titoli dell’*header* della *dashboard*, *titles*, è stato dotato della proprietà ’/magazine’. La stringa ’Magazine’, il suo valore, diverrà il titolo della sezione “Magazine”.

```

1 const titles = {
2   // [...]
3   '/magazine': 'Magazine',
4   // [...]
5 };

```

Listing 5.18: Aggiornamento variabile *titles*

In secondo luogo, come mostrato del Listing 5.19, si è intervenuto sulla variabile che contiene la lista dei componenti che formano il menu laterale, *sidebar*, alla quale è stato aggiunto quello

relativo alla sezione “Magazine”. Le *props* del nuovo elemento Sidebar appena introdotto sono:

- **label**: stabilisce il testo dell’etichetta della sezione nel menu laterale, il suo valore è la stringa ‘Magazine’;
- **href**: stabilisce il percorso da seguire per visualizzare il componente, il suo valore è la stringa ‘/magazine’;
- **icon**: stabilisce l’immagine da visualizzare accanto all’etichetta, il suo valore è il componente `NewspaperIcon`.

```

1 const sidebar = (
2   <List>
3     {/* [...] */}
4     <SidebarItem
5       label="Magazine" href="/magazine"
6       icon={<NewspaperIcon />} isActive={isActive}
7     />
8     {/* [...] */}
9   </List>
10 );

```

Listing 5.19: Aggiornamento variabile sidebar

Il **terzo passo** ha avuto a che fare con la creazione delle funzioni `getMagazines()`, `createMagazine()` e `deleteMagazine()`, le funzioni che effettuano le chiamate alle API visibili nel Listing 5.20. All’interno del loro corpo (alle righe righe 1, 4 e 7) sono state utilizzate, rispettivamente, le funzioni `get()`, `post()` e `del()`, funzioni che hanno il compito di effettuare le richieste avvalendosi dei metodi GET, POST e DELETE.

```

1 export const getMagazines = () => get('/magazine', { baseUrl: '',
2   facilities });
3
3 export const createMagazine = body =>
4   post('/magazine', body, { baseUrl: 'facilities' });
5
6 export const deleteMagazine = uuid =>

```

```
7 del('/magazine/${uuid}', undefined, { baseUrl: 'facilities' }) ;
```

Listing 5.20: Funzioni di chiamata alle API per la sezione “Magazine”

Il **quarto e ultimo passo**, il più complesso e lungo, ha riguardato la modellazione dei componenti che formano la sezione “Magazine” e con i quali è possibile interagire.

PageMagazines, il primo realizzato (la cui istanza è stata vista nel *routing* del ruolo utente *marketing admin*), è illustrato nel Listing 5.21 ed ha lo scopo di effettuare il *rendering* della tabella che contiene la lista dei magazine.

```
1 export default function PageMagazines() {
2   // [...]
3   return (
4     // [...]
5   );
6 };
```

Listing 5.21: Componente PageMagazines

Innanzitutto, al componente è stata fornita la logica necessaria per il suo funzionamento.

Come mostrato dal Listing 5.22, la funzione `getMagazines()` è passata come argomento all’*hook* `useApi()` per reperire i magazine da visualizzare, `magazines`, le informazioni relative al caricamento e all’errore, `isLoading` ed `error`, e la funzione che permette di aggiornare la lista dei magazine, `fetchMagazines`.

```
1 const {
2   data: magazines,
3   isLoading,
4   error,
5   silent: fetchMagazines,
6 } = useApi(getMagazines, [], []);
```

Listing 5.22: Utilizzo `getMagazines()`

I due *handler* dichiarati in `PageMagazines` sono `handleDeleteMagazine()` e `handleConfirmDeleteMagazine`. Come mostrato nel Listing 5.23, se da un lato, il primo è noto per struttura e scopo poiché già incontrato nella sezione 5.3, dall’altro, il secondo costituisce un qualcosa di completamente nuovo.

La funzione `handleDeleteMagazine`, infatti, appartiene a quel gruppo di *handler* che hanno il compito di aprire una modale, e le operazioni che esegue sono:

1. L'impostazione, tramite il *setter* `setData()`, dei dati della modale `modalConfirmDeleteMagazine`, sull'oggetto `magazine`;
2. L'apertura della modale tramite la funzione `open()`.

Gli oggetti relativi alla modale `modalConfirmDeleteMagazine` e `modalMagazine` sono stati dichiarati attraverso il *custom hook* `useModal()` (alle righe riga 1 e 3); `modalMagazine` è utilizzata per l'aggiunta di nuovi magazine.

L'*handler* `handleConfirmDeleteMagazine`, invece, è una funzione atta all'effettiva manipolazione dei dati, e le operazioni che esegue sono:

1. L'eliminazione, dal *bucket marketing*, della proprietà `path` dell'oggetto 0 all'interno dell'*array files* (contenuto a sua volta, a risalire, dagli oggetti `document`, `magazine`, `data` e `modalConfirmDeleteMagazine`), con l'invocazione della funzione `removeToS3()`;
2. L'eliminazione, dal *bucket marketing*, della proprietà `path` contenuta dall'oggetto `image` (contenuta a sua volta, a risalire, da `magazine`, `data` e `modalConfirmDeleteMagazine`), con l'invocazione della funzione `removeToS3()`;
3. L'effettiva eliminazione del magazine, con l'invocazione della funzione `deleteMagazine()`, alla quale viene passato come argomento l'`uuid` dell'oggetto `magazine` (contenuto a sua volta, a risalire, dagli oggetti `data` e `modalConfirmDeleteMagazine`);
4. L'aggiornamento della lista dei magazine, con l'invocazione della funzione `fetchMagazines()`.

```

1 const modalMagazine = useModal();
2
3 const modalConfirmDeleteMagazine = useModal();
4
5 const handleDeleteMagazine = magazine => {
6   modalConfirmDeleteMagazine.setData({ magazine });
7   modalConfirmDeleteMagazine.open();
8 };
9
10 const handleConfirmDeleteMagazine = async () => {

```

```

11  await removeToS3(
12    'marketing',
13    modalConfirmDeleteMagazine.data.magazine.document.files[0].path
14  );
15
16  await removeToS3(
17    'marketing',
18    modalConfirmDeleteMagazine.data.magazine.image.path
19  );
20
21  await deleteMagazine(modalConfirmDeleteMagazine.data.magazine.uuid);
22
23  fetchMagazines();
24 };

```

Listing 5.23: *Handler* componente PageMagazines

PageMagazines contiene poi la logica relativa al campo di ricerca. Il suo funzionamento è sostanzialmente identico a quello visto nella sezione 5.4, ma la sua struttura si differenzia parzialmente da quella della *feature* F5.

La dinamica di filtraggio è qui inserita all'interno dell'*hook* `useEffect()`; inoltre, la condizione `if` (a riga 9) si caratterizza per il fatto che il valore/i restituito dal filtraggio<sup>15</sup>, effettuato nuovamente tramite l'iteratore `filter()`, corrisponde all'elemento/i, all'interno dei magazine, il cui titolo in minuscolo corrisponde a quando presente nella variabile `_filterName`.

```

1 const [filtered, setFiltered] = useState([]);
2
3 const [filterName, setFilterName] = useState('');
4
5 useEffect(() => {
6   let _filtered = [...magazines];
7   let _filterName = filterName.toLowerCase().trim();
8
9   if (_filterName.length > 0) {
10     _filtered = _filtered.filter(
11       el => el.title.toLowerCase().indexOf(_filterName) > -1
12     );

```

---

<sup>15</sup>Il valore restituito dal filtraggio è assegnato alla variabile `_filtered`.

```

13     }
14
15     setFiltered(_filtered);
16 }, [filterName, magazines]);
17
18 const filter = [filterName.length > 0 && 'titolo'].filter(el => el).join
  ('', '');

```

Listing 5.24: Dinamica di filtraggio della sezione magazine

La logica del primo componente creato si è conclusa specificando la posizione del *bucket* in cui sono contenute le immagini associate ai magazine. Ciò è stato fatto grazie alla funzione `configure()` (invocata sull'oggetto Amplify), alla quale è passato come argomento un oggetto la cui unica proprietà, Storage, ha per valore l'ubicazione, ovvero la proprietà `marketing` di `storage` (contenuta, a risalire, dagli oggetti `amplify` e `config`).

```

1 Amplify.configure({
2   Storage: config.amplify.storage.marketing,
3 });

```

Listing 5.25: Configurazione di Amplify

Dopodiché, è stata modellata la UI di PageMagazines. Il *rendering* di questo componente avviene in modo particolare: all'interno della sua logica, è dichiarato un oggetto, `table`, che specifica tutte le caratteristiche della tabella da visualizzare, e che viene passato, sotto forma di `prop`, a un componente, `SuperTable`, che “impagina” `table` ritornando a schermo la tabella dei magazine.

Per questo motivo, la gestione della veste grafica di PageMagazines è consistita principalmente nella manipolazione dell'oggetto `table`<sup>16</sup>, le cui proprietà sono:

- `title` (a riga 2): specifica il titolo della tabella, il suo valore è la stringa `'Magazine'`;
- `noResults` (a riga 2): specifica il messaggio da restituire se la tabella è vuota, il suo valore è la stringa `'Nessun magazine trovato'`;
- `action` (da riga 3 a 31): specifica le azioni che possono essere compiute nell'*header* della tabella, il suo valore è un'espressione che permette di effettuare:

<sup>16</sup>L'oggetto `table` deve essere costruito necessariamente includendo proprietà che possono essere accettate dal componente `SuperTable`.

- La ricerca di un magazine fra quelli esistenti grazie al componente `TextField` (da riga 5 a 22). Le molteplici *props* di cui è dotato sono sostanzialmente identiche a quelle descritte in relazione al medesimo componente nella *feature* F5<sup>17</sup>;
- L’aggiunta di un nuovo magazine grazie al componente `Button` (da riga 24 a 29). Il testo restituito dal componente è “Magazine” e le sue più importanti *props* sono:
  - \* `onClick` (a riga 25): il suo valore è la funzione `open()` invocata sull’oggetto `modalMagazine`;
  - \* `startIcon` (a riga 26): il suo valore è il componente `AddIcon`;
- `isLoading` e `error` (a riga 32): specificano delle informazioni relative allo stato di una chiamata ad un’API;
- `filter` (a riga 32): specifica il messaggio da ritornare sotto l’*header* della tabella una volta eseguita una ricerca;
- `perPage` (a riga 32): specifica il numero di elementi da visualizzare in una pagina, il suo valore è 10;
- `headers` (da riga 33 a 40): specifica i titoli delle colonne della tabella, il suo valore è un’oggetto che, a sua volta, include gli oggetti `title`, `description`, `date`, `image`, `document` e `delete`. All’interno di ognuno di essi sono definite le caratteristiche del titolo stesso attraverso le proprietà:
  - `label`: stabilisce il testo del titolo di una colonna, il suo valore è una stringa, ad esempio, ’Titolo’;
  - `sort`: stabilisce se è possibile effettuare l’ordinamento dei magazine in base alla proprietà della colonna in considerazione, il suo valore è un valore booleano, ad esempio, `true`;
  - `width`: stabilisce la larghezza della colonna, il suo valore è un numero, ad esempio, 40;
- `rows` (da riga 41 a 57): specifica il contenuto delle righe della tabella, il suo valore è il *mapping* dell’*array* `filtered`. Da un lato, l’*array* `filtered` permette di visualizzare, ogni qualvolta viene effettuata una ricerca, solamente le righe della tabella che corrispondono a

<sup>17</sup>Si veda la sezione 5.4.

quanto cercato; dall’altro, se non viene effettuata nessuna ricerca, consente semplicemente di effettuare il *rendering* della lista dei magazine.

Tramite il *mapping*, le proprietà di cui ogni elemento dell’*array* è fornito sono:

- **key** (a riga 42): il suo valore è la proprietà `uuid` dell’oggetto `e1`;
- **title** (a riga 42): specifica il titolo, il suo valore è la proprietà `title` dell’oggetto `e1`;
- **description** (a riga 43): specifica la descrizione, il suo valore è il componente `Box`. Questo permette di visualizzare la descrizione, originariamente in formato HTML, come normale testo, mantenendo la formattazione fornitagli lato client. Ciò avviene grazie alla *prop* `dangerouslySetInnerHTML` passata al componente `Box`: il suo valore è un oggetto la cui unica proprietà, `__html`, ha come valore la descrizione del magazine, ovvero la proprietà `description` dell’oggetto `e1`;
- **date** (a riga 44): specifica la data, il suo valore è la proprietà `date` dell’oggetto `e1`, la quale viene formattata con il metodo `format()`<sup>18</sup>;
- **image** (a riga 45): specifica l’immagine, il suo valore è l’istanza del componente `Image`. L’unica *prop* che le viene passata è `e1`, il cui valore è quello del singolo oggetto che sta venendo mappato, `e1`;
- **document** (a riga 46): specifica il documento, il suo valore è il componente `DownloadDocument`. Le *props* passategli sono:
  - \* `action`: stabilisce il testo visualizzato dal componente. Il suo valore è la proprietà `name` (contenuta, a risalire, dagli oggetti `document` e `e1`);
  - \* `storage`: stabilisce la posizione del documento per effettuarne il download, il suo valore è la stringa ‘Marketing’;
  - \* `path`: stabilisce il percorso per reperire il documento per effettuarne il download, il suo valore è la proprietà `path` contenuta nell’oggetto 0 dell’*array* `files` (contenuto, a sua volta, dagli oggetti `document` ed `e1`).
- **delete** (a riga 52): specifica il bottone per l’eliminazione di un magazine, il suo valore è il componente `IconButton`. L’unica *prop* fornитagli è `onClick`, il cui valore è l’*handler* `handleDeleteMagazine()` a cui viene passato come argomento `e1`, l’og-

---

<sup>18</sup>Si veda la sottosezione 4.3.4

getto del magazine che si vuole eliminare. All'interno di IconButton è presente il componente DeleteIcon.

```
1 const table = {
2   title: 'Magazine', noResults: 'Nessun magazine trovato',
3   action: (
4     <>
5       <TextField
6         sx={{ width: 300, mr: 2 }}
7         size="small" variant="filled" label="Filtra per nome"
8         placeholder="Cerca..." value={filterName}
9         onChange={e => setFilterName(e.target.value)}
10        InputProps={{
11          endAdornment: (
12            <InputAdornment
13              position="end"
14              style={{ display: filterName.length > 0 ? 'flex' : 'none' }}>
15            <IconButton onClick={() => setFilterName('')}>
16              <ClearIcon />
17            </IconButton>
18            </InputAdornment>
19          ) ,
20        } }
21      } }
22    />
23
24    <Button
25      variant="contained" onClick={modalMagazine.open}
26      startIcon={<AddIcon />}>
27    >
28      Magazine
29    </Button>
30  </>
31  ) ,
32  isLoading, error, filter, perPage: 10,
33  headers: {
34    title: { label: 'Titolo', sort: false },
```

```

35   description: { label: 'Descrizione', sort: false },
36   date: { label: 'Data' },
37   image: { label: 'Immagine', sort: false },
38   document: { label: 'Documento', sort: false },
39   delete: { label: '', sort: false, width: 40 },
40 },
41 rows: filtered.map(el => (
42   key: el.uuid, title: el.title,
43   description: <Box dangerouslySetInnerHTML={{ __html: el.description
44 }} />,
45   date: format(new Date(el.date), 'dd/MM/yyyy'),
46   image: <Image el={el} />,
47   document: (
48     <DownloadDocument
49       action={el.document.name} storage="marketing"
50       path={el.document.files[0].path}
51     />
52   ),
53   delete: (
54     <IconButton onClick={() => handleDeleteMagazine(el)}>
55       <DeleteIcon fontSize="small" />
56     </IconButton>
57   ),
58 ))),
59 };

```

Listing 5.26: Oggetto table

Il Listing 5.27 mostra come il *rendering* della tabella avvenga tramite il componente SuperTable (a riga 2), al quale viene passata come unica *prop* table, il cui valore è l’oggetto table appena discusso.

Al suo interno, è mostrato anche come le modalì per l’aggiunta e l’eliminazione di un magazine vengano restituite condizionalmente attraverso l’operatore logico AND.

Nel primo caso, se la proprietà isOpen dell’oggetto modalMagazine è vera, allora viene restituito il componente ModalMagazine; le *props* passategli sono:

- modal: stabilisce la modale in uso, il suo valore è l’oggetto modalMagazine;
- magazine: stabilisce le proprietà del magazine, il suo valore è un oggetto vuoto ({});

- `onSaved`: stabilisce la funzione da invocare per l'aggiornamento della lista dei magazine, il suo valore è la funzione `fetchMagazines()`.

Allo stesso modo, nel secondo caso, se la proprietà `isOpen` dell'oggetto `modalConfirmDeleteMagazine` è vera, allora viene restituito il componente `ModalConfirm`; le *props* passategli sono:

- `modal`: il suo valore è l'oggetto `modalConfirmDeleteMagazine`;
- `title`: stabilisce il titolo della modale di conferma eliminazione, il suo valore è la stringa `"Eliminazione magazine"`;
- `text`: stabilisce il testo del messaggio mostrato dalla modale di conferma eliminazione, il suo valore è la stringa `"Eliminare il magazine?"`;
- `handleConfirm`: stabilisce la funzione da invocare per l'effettiva eliminazione del magazine.

```

1 <Box p={3}>
2   <SuperTable table={table} />
3
4   {modalMagazine.isOpen && (
5     <ModalMagazine
6       modal={modalMagazine} magazine={[]} onSaved={fetchMagazines}
7     />
8   )}
9
10  {modalConfirmDeleteMagazine.isOpen && (
11    <ModalConfirm
12      modal={modalConfirmDeleteMagazine}
13      title="Eliminazione magazine"
14      text="Eliminare il magazine?"
15      handleConfirm={handleConfirmDeleteMagazine}
16    />
17  )}
18 </Box>;

```

Listing 5.27: SuperTable e modali

`Image`, il secondo componente creato (la cui istanza è stata vista nella UI di `PageMagazines`), ha il solo compito di reperire l’immagine di un magazine.

Come mostrato dal Listing 5.28, la sua dichiarazione è stata accompagnata dall’importazione della `prop el`, l’unica che gli viene passata durante il *mapping* effettuato nella proprietà `rows` dell’oggetto `table`.

```

1 const Image = ({ el }) => {
2   // [...]
3   return (
4     // [...]
5   )
6 }
```

Listing 5.28: Componente `Image`

La logica di `Image` è molto semplice. Da un lato, come possibile vedere nel Listing 5.29, sono stati dichiarati la variabile di stato `image`, che ha il compito di “memorizzare” l’immagine da reperire, ed il relativo *setter* `setImage()`; la variabile di stato è stata inizializzata con il valore `null`.

```
1 const [image, setImage] = useState(null);
```

Listing 5.29: Variabile di stato `image` di `Image`

Dall’altro, come illustrato dal Listing 5.30, l’*hook* `useEffect()` consente di reperire l’immagine. All’interno del suo corpo, ciò avviene grazie a una condizione che utilizza un singolo `if` (da riga 2 a 6) per esprimere il seguente concetto: se l’oggetto `el` è vero, allora recupera la proprietà `path` dell’oggetto `image`(contenuta, a risalire, dall’oggetto `el`), tramite la funzione `get()` (invocata sull’oggetto `Storage`); in seguito, sostituisce il precedente valore della variabile di stato `image` con `url`, grazie al *setter* `setImage()`,

```

1 useEffect(() => {
2   if (el) {
3     Storage.get(el.image.path).then(url => {
4       setImage(url);
5     });
6   }
}
```

```
7 } , [el]);
```

Listing 5.30: `useEffect()` di `Image`

Invece, la UI di `Image` è costituita solamente dal componente `CardMedia`, le cui più importanti *props* sono:

- `image`: il suo valore è `image`;
- `alt`: stabilisce qual è il testo alternativo, il suo valore è la proprietà `image` dell'oggetto `el`.

```
1 <CardMedia
2   component="img"
3   height="200"
4   image={image}
5   alt={el.image}
6 />;
```

Listing 5.31: UI `Image`

Infine, `ModalMagazine` (la cui istanza è stata vista nel *rendering* condizionale delle modali, a seguito di `SuperTable`) è il terzo e ultimo componente creato per la sezione “Magazine”; il suo compito è la creazione di un nuovo magazine. Come mostrato dal Listing 5.32, la sua dichiarazione è stata accompagnata dall’importazione delle *prop* `modal` e `onSaved`.

```
1 export default function ModalMagazine({ modal, onSaved }) {
2   // [...]
3   return (
4     // [...]
5   )
6 }
```

Listing 5.32: Componente `ModalMagazine`

Anche in questo caso, per prima cosa è stata definita la logica del componente. Come mostrato dal Listing 5.33, all’interno del componente `ModalMagazine`, per ogni proprietà posseduta da un magazine è stata dichiarata una variabile di stato e il relativo *setter* tramite l’*hook* `useState()`. Il compito di queste variabili è la “memorizzazione” dei valori immessi, lato

client, durante l'aggiunta di un nuovo magazine. Il valore con cui sono state inizializzate differisce a seconda del tipo di dato atteso: `title`, `description`, `image_alt` e `doc_name` hanno come valore di inizio una stringa vuota (''), mentre `date`, `image` e `doc` hanno `null`.

```

1 const [title, setTitle] = useState('');
2 const [description, setDescription] = useState('');
3 const [date, setDate] = useState(null);
4 const [image_alt, setImageAlt] = useState('');
5 const [image, setImage] = useState(null);
6 const [doc_name, setDocName] = useState('');
7 const [doc, setDoc] = useState(null);

```

Listing 5.33: Variabili di stato di ModalMagazine

In seguito, come mostrato dal Listing 5.34, è stato definito l'oggetto `schema`. Al suo interno, le proprietà, anche in questo caso, riflettono quelle possedute dal magazine da aggiungere e il loro valore riflette i vincoli che devono essere rispettati dall'utente al momento di immissione dei dati.

```

1 const schema = yup.object().shape({
2   title: yup.string().required(),
3   description: yup.string().required(),
4   date: yup.date().typeError('Data non valida').required(),
5   image_alt: yup.string().required(),
6   image: yup.mixed().nullable(true).required("Seleziona un'immagine"),
7   doc_name: yup.string().required(),
8   doc: yup.mixed().nullable(true).required('Seleziona un documento'),
9 });

```

Listing 5.34: Oggetto shema di ModalMagazine

Come illustrato dal Listing 5.35, inoltre, dentro a `ModalMagazine` sono stati effettuati:

- Il “recupero” delle funzioni `validate()`, `validateField()`, `validation()` e `validationProps()` con l’*hook* `useValidation()` (a riga 1 e 2);
- La dichiarazione dell’oggetto `status` con l’*hook* `useStatus()` (a riga 4);
- La dichiarazione dell’oggetto `descriptionRef` con l’*hook* `useRef()` (a riga 5);

```

1 const { validate, validateField, validation, validationProps } =
2   useValidation(schema);
3
4 const status = useStatus();
5
6 const descriptionRef = useRef();

```

Listing 5.35: *Custom hooks* di ModalMagazine

La parte più importante della logica di ModalMagazine è quella riguardante l'*handler onClick*, funzione che (come `handleConfirmDeleteMagazine`) si occupa dell'effettiva manipolazione di dati. Come possibile vedere nel Listing 5.36, le operazioni che esegue sono:

1. La dichiarazione dell'oggetto `form`, le cui proprietà sono le variabili di stato definite in precedenza (da riga 2 a 4);
2. La valutazione di una condizione `if` (da riga 6 a 38) che stabilisce che, se la validazione dell'oggetto `form`, tramite l'invocazione della funzione `validate()`, è avvenuta correttamente, può essere eseguita un'altra serie di operazioni. Da un lato, fra di esse vi è il tentativo, `try` (da riga 7 a 35), di eseguire le seguenti istruzioni:
  - (a) L'impostazione dello stato su “caricamento” con l'invocazione del metodo `setLoading()` (sull'oggetto `status`, a riga 8);
  - (b) La dichiarazione della variabile `import_image`, che carica l'immagine, `image`, e il suo testo alternativo, `image_alt`, all'interno del *bucket* `marketing` (da riga 10 a 12);
  - (c) La dichiarazione della variabile `import_doc`, che carica il documento, `doc`, e il suo nome, `doc_name`, all'interno del *bucket* `marketing` (da riga 14 a 16);
  - (d) La dichiarazione dell'oggetto `body`, le cui proprietà sono le variabili di stato definite precedentemente a eccezione di `description`, `image` e `document`. I valori a esse fornite sono, rispettivamente, il valore ritornato dalla conversione del contenuto della variabile di stato `description`, con l'invocazione della funzione `htmlFromRaw()`; tutte le proprietà dell'oggetto `file` (contenuto dall'oggetto `import_image`) uguali tranne `alt`, a cui viene assegnato il contenuto della variabile di stato `image_alt`; la proprietà `date` e le proprietà `name` e `file` a cui sono assegnati, rispettivamente, i valori delle variabili di stato `doc_name` e l'oggetto `file` (contenuto dall'oggetto `import_doc`) (da riga 18 a 26);

- (e) L'effettiva creazione del nuovo magazine con l'invocazione della funzione di chiamata all'API `createMagazine()`; i dati del nuovo magazine saranno quelli contenuti nell'oggetto `body` creato poco fa, il quale viene passato come argomento alla funzione `createMagazine()` (a riga 28).
- (f) L'impostazione dello stato su "successo" con l'invocazione della funzione `setStatus()` (sull'oggetto `status`) (a riga 30);
- (g) L'aggiornamento della lista dei magazine con l'invocazione della funzione `onSaved()`. Il suo valore è la funzione `fetchMagazines`, la quale era stata passata al componente `modalMagazine` come valore della *prop* `onSaved` (a riga 32);
- (h) La chiusura della modale con l'invocazione della funzione `close()` (sull'oggetto `modal`) (a riga 34).

Dall'altro lato, fra le operazioni previste dalla condizione `if`, vi è il recupero, `catch`, dell'errore, `err`, e la conseguente impostazione dello stato su "errore" con l'invocazione della funzione `setError()` (sull'oggetto `status`) (da riga 35 a 37).

```

1 const onClick = async () => {
2   const form = {
3     title, description, date, image_alt, image, doc_name, doc,
4   };
5
6   if (await validate(form)) {
7     try {
8       status.setLoading();
9
10      const import_image = await uploadToS3(
11        'marketing', image, image_alt, 'magazine',
12      );
13
14      const import_doc = await uploadToS3(
15        'marketing', doc, doc_name, 'magazine',
16      );
17
18      const body = {
19        title,
20        description: htmlFromRaw(description),

```

```

21     date ,
22     image: { ...import_image.file, alt: image_alt },
23     document: {
24       name: doc_name, date, file: import_doc.file,
25     },
26   };
27
28   await createMagazine(body);
29
30   status.setSuccess();
31
32   onSave();
33
34   modal.close();
35 } catch (err) {
36   status.setError(err);
37 }
38 }
39 };

```

Listing 5.36: *Handler onClick()* di ModalMagazine

Infine, è stata realizzata la complessa UI di ModalMagazine, operazione che conclude il processo di implementazione della sezione “Magazine”. Tutti gli elementi che la compongono sono contenuti all’interno del componente padre Dialog, il quale si occupa del *rendering* l’intera modale; le sue più importanti *props* sono:

- open: il suo valore è la proprietà `isOpen` dell’oggetto `modal`;
- onClose: il suo valore è la funzione `close()` dell’oggetto `modal`;

La struttura di Dialog è formata dai componenti figli `DialogTitle`, `DialogContent`, `Error` e `DialogActions`.

`DialogTitle` si occupa del *rendering* del titolo della modale, “Magazine”, e del bottone per di chiusura della modale, il componente `ModalCloseButton`. A quest’ultimo è passata solamente la *prop* `modal` a cui è assegnato come valore l’oggetto `modal`.

`DialogContent` ha invece come compito il *rendering* del corpo della modale; al suo interno sono presenti due grandi componenti `Grid` padre che delimitano le due aree in cui la modale si divide, una dedicata alla gestione dell’immagine, e una a quella del documento.

Ognuno dei Grid racchiude a sua volta tanti elementi figli Grid quanti sono i campi previsti dall’immagine o dal documento, e ognuno dei Grid figli contiene, a sua volta, gli effettivi componenti atti alla gestione di uno specifico campo. Nel caso dell’immagine nel caso dell’immagine sono:

- `TextField` (da riga 12 a 16): permette di immettere il titolo del magazine da aggiungere.

Le *props* più importanti di cui è dotato sono:

- `label` e `placeholder`: il loro valore è la stringa ‘Titolo’;
- `value`: il suo valore è la variabile di stato `title`.

Inoltre, è presente un’espressione atta alla validazione del testo immesso e, quando questa ha un’esito positivo, al suo “salvataggio”. Ciò vien fatto passando il campo da validare, ‘`title`’ e il *setter* `setTitle()` come argomenti della funzione `validationProps()`;

- `Typography` (da riga 20 a 22): effettua il *rendering* del sottotitolo “Descrizione”;
- `MUIRichTextEditor`: permette di immettere e di formattare il testo della descrizione;
- `FormHelperText`: effettua il *rendering* del messaggio di errore se la validazione non è andata a buon fine
- `DatePicker`: permette di scegliere una data. Le sue *props* più importanti sono:
  - `label`: il suo valore è la stringa ‘Data’;
  - `value`: il suo valore è la variabile di stato `date`;
  - `onChange`: il suo valore è una funzione che esegue due operazioni: da un lato la sostituzione del precedente valore della variabile di stato `date` grazie all’utilizzo del *setter*  `setDate()`, dall’altro la validazione della data inserita con il passaggio della stringa ‘`date`’ e della variabile di stato `date` come argomenti della funzione `validate()`;
  - `renderInput`: il suo valore è una funzione che permette la personalizzazione del componente per la scelta della data attraverso l’elemento `TextField`.
- A differenza di quanto visto per i precedenti componenti, `Typography`, `TextField`, `Upload` e `FormHelperText` sono contenuti in un unico Grid:
  - `Typography`: effettua il *rendering* del sottotitolo dell’area dedicata all’immagine, ovvero “Immagine”;

- Typography: effettua il *rendering* del testo che fornisce indicazioni sul campo, ovvero “Alt immagine”;
- TextField: permette l’inserimento del titolo alternativo. Il valore della sua *prop* più importante, `value`, è la variabile di stato `image_alt`; di nuovo, è presente un’espressione atta alla validazione del testo immesso e del suo “salvataggio”. In questo caso, gli argomenti forniti a `validationProps()` sono `'image_alt'` e `setImageAlt`; Inoltre, è presente un’espressione atta alla validazione del testo immesso e, quando questa ha un’esito positivo, al suo “salvataggio”. Ciò vien fatto passando il campo da `validare`, `'title'` e il *setter* `setTitle()` come argomenti della funzione `validationProps()`;
- Typography: effettua il *rendering* del testo che fornisce indicazioni sul campo, ovvero “Carica immagine”;
- Upload: permette l’aggiunta di un’immagine. Le sue *props* più importanti sono:
  - \* `file`: stabilisce il file che viene caricato, il suo valore è la variabile di stato `image`;
  - \* `setFile`: stabilisce la funzione per il caricamento del file, il suo valore è il *setter* `setFile()`;

Il componente `Error` è posizionato subito dopo il corpo della tabella ed ha lo scopo di ritornare un messaggio di errore. L’unica *prop* di cui è dotato è `error`, il cui valore è la proprietà `error` dell’oggetto `status`.

L’ultimo componente della modale è `DialogActions`, elemento che contiene i bottoni:

- Button: effettua il *rendering* del testo “Annulla” e permette la chiusura della modale. La sua unica *prop* è `onClick`, il suo valore è la funzione `close()` dell’oggetto `modal`;
- LoadingButton: effettua il *rendering* del testo “Conferma” e il salvataggio dei dati immessi lato client per il magazine da aggiungere. Le sue *props* più importanti sono:
  - `onClick`: il suo valore è l’*handler* `onClick()`;
  - `loading`: il suo valore è la proprietà `isLoading` dell’oggetto `status`.

```

1 <Dialog
2   open={modal.isOpen} onClose={modal.close}
3   scroll="body" fullWidth maxWidth="xs"

```

```
4 >
5   <DialogTitle>
6     Magazine <ModalCloseButton modal={modal} />
7   </DialogTitle>
8
9   <DialogContent>
10    <Grid container spacing={1}>
11      <Grid item xs={12}>
12        <TextField
13          required fullWidth variant="filled"
14          label="Titolo" placeholder="Titolo" value={title}
15          {...validationProps('title', setTitle)}
16        />
17      </Grid>
18
19      <Grid item xs={12}>
20        <Typography variant="subtitle2" sx={{ mt: 1 }}>
21          Descrizione
22        </Typography>
23      </Grid>
24
25      <Grid item xs={12}>
26        <Paper variant="outlined" sx={{ mb: 1 }}>
27          <MUIRichTextEditor
28            {...getEditorParams(
29              descriptionRef, description, setDescription
30            )}
31            controls={defaultControlsWithLink}
32          />
33        </Paper>
34
35        {!validation.description.valid && (
36          <FormHelperText
37            style={{ marginTop: '.5rem' }}
38            error={!validation.description.valid}
39            >{validation.description.error}</FormHelperText>
40        )}
41      </Grid>
42
```

```
43      <Grid item xs={12}>
44        <DatePicker
45          label="Data" value={date}
46          onChange={date => {
47            setDate(date);
48            validateField('date', date);
49          }}
50          renderInput={params => (
51            <TextField
52              {...params} fullWidth required variant="filled"
53              {...validationProps('date')}
54            />
55          )}
56        />
57      </Grid>
58
59      <Grid item xs={12}>
60        <Typography variant="subtitle2" sx={{ mt: 2 }}>
61          Immagine
62        </Typography>
63        <Typography variant="body2" sx={{ mt: 1 }}>
64          Alt immagine
65        </Typography>
66        <TextField
67          sx={{ my: 1 }}
68          required fullWidth variant="filled"
69          label="Alt immagine" placeholder="Alt immagine"
70          value={image_alt}
71          {...validationProps('image_alt', setImageAlt)}
72        />
73        <Typography variant="body2" sx={{ mt: 1 }}>
74          Carica immagine
75        </Typography>
76        <Upload
77          file={image} setFile={setImage}
78          label="Trascina qui l'immagine"
79        />
80        {!validation.image.valid && (
81          <FormHelperText
```

```

82         style={{ marginTop: '.5rem' }}
83         error={!validation.image.valid}
84       >{validation.image.error}</FormHelperText>
85     )
86   </Grid>
87
88   <Grid>
89     {/* [...] */}
90   </Grid>
91 </Grid>
92 </DialogContent>
93
94 <Error error={status.error} />
95
96 <DialogActions>
97   <Button onClick={modal.close}>Annulla</Button>
98
99   <LoadingButton onClick={onClick} loading={status.isLoading}>
100    Conferma
101  </LoadingButton>
102 </DialogActions>
103 </Dialog>

```

Listing 5.37: UI ModalMagazine

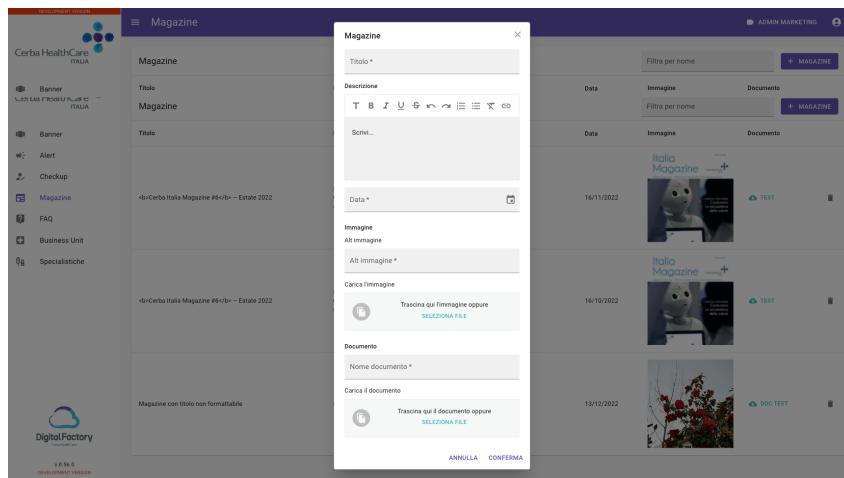
***Output grafico***

Figura 5.13: Modale aggiunta nuovo magazine

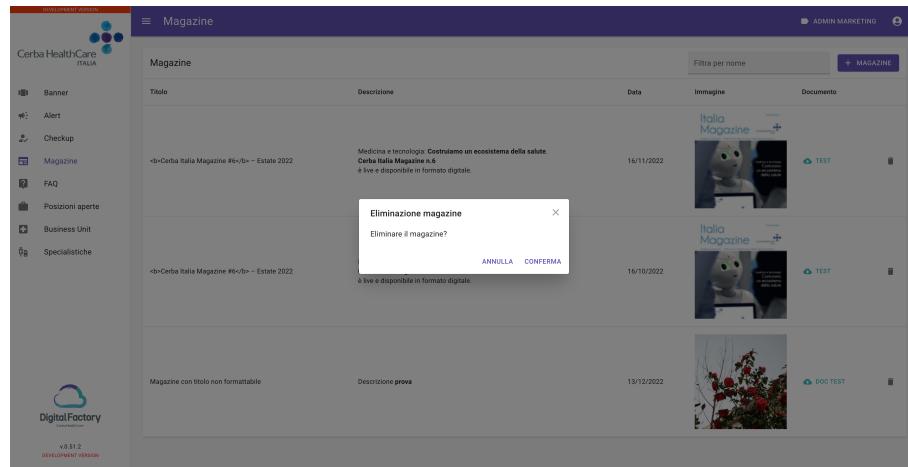


Figura 5.14: Modale eliminazione magazine esistente

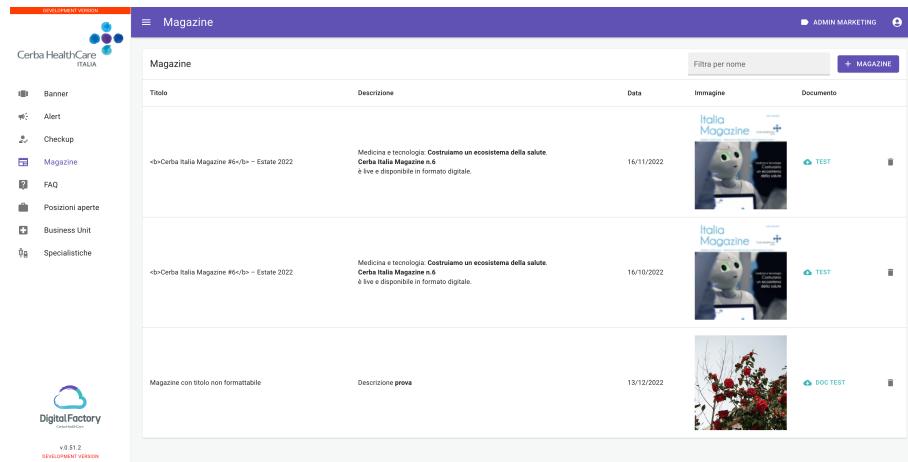


Figura 5.15: Tabella magazine campo ricerca vuoto

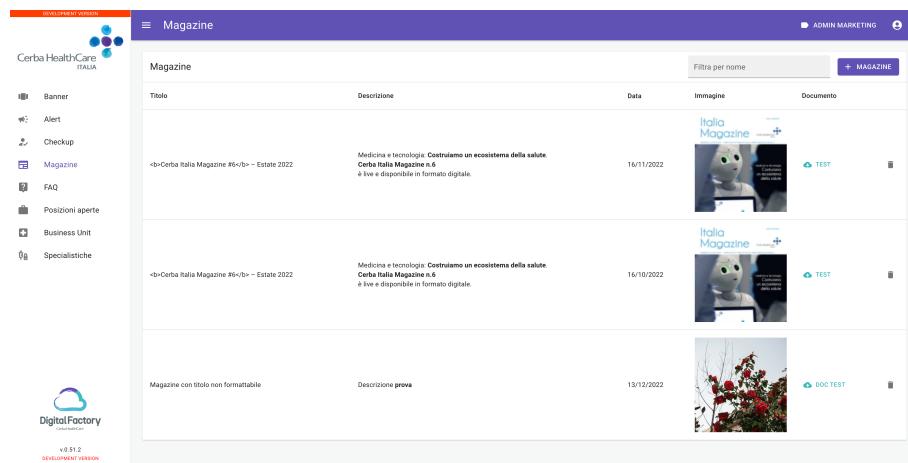


Figura 5.16: Tabella magazine campo ricerca riempito

### 5.5.2 F7: “FAQ”

**Requisiti** Aggiungere la sezione “FAQ” fra quelle previste per il ruolo utente *marketing admin*.

Al suo interno, una tabella deve mostrare le FAQ esistenti in una lista semplice e riportarne le proprietà “Domanda”, “Risposta” e “Categoria”.

Deve essere possibile aggiungere, modificare o eliminare una FAQ.

Solamente per la proprietà “Risposta” deve essere fornita la possibilità di formattare il testo.

L’*header* della tabella delle FAQ deve essere dotato di un campo di ricerca. Questo deve essere in grado di filtrare per domanda e posizionato alla sinistra del bottone per l’aggiunta di una nuova FAQ.

### 5.5.3 F8: “Business Unit”

**Requisiti** Aggiungere la sezione “Business Unit” fra quelle previste per il ruolo utente *marketing admin*.

Al suo interno, una tabella deve mostrare le *business unit* (BU) esistenti in una lista semplice e riportarne la proprietà “Nome”.

L’*header* della tabella delle BU deve essere dotato di un campo di ricerca. Questo deve essere in grado di filtrare per nome e posizionato all’estrema destra dell’*header* della tabella.

Ogni BU deve essere dotata di una pagina di dettaglio alla quale si accede cliccando sul link posto in concomitanza del nome. Qui, deve essere possibile modificare e visualizzare la descrizione, campo che deve poter essere formattato; sempre all’interno della pagina di dettaglio, si devono poter aggiungere, visualizzare ed eliminare un’immagine e un’icona.

## Output grafico

The screenshot shows a user interface for managing Business Units. On the left, there's a sidebar with icons for Banner, Alert, Checkup, Magazine, FAQ, Business Unit, and Specialistice. The main area has a purple header 'Business Unit'. Below it is a table with a single row containing the word 'Radiologia'. At the top right of the table is a search bar labeled 'Filtra per nome'.

Figura 5.17: Tabella BU campo ricerca vuoto

This screenshot shows the same interface as Figure 5.17, but with a search term 'ta' entered into the search bar. The table now displays a single row for 'Radiologia', and a message above the table states '1 riga trovata filtrando per: nome'.

Figura 5.18: Tabella BU campo ricerca riempito

This screenshot shows the detailed view of a selected Business Unit. The top navigation bar includes 'Business Unit / Radiologia'. The main content area shows the name 'Radiologia' and a 'Descrizione' section. To the right, there are buttons for 'MODIFICA', 'Immagine' (with '+ AGGIUNGI'), and 'Icona' (with '+ AGGIUNGI').

Figura 5.19: Pagina dettaglio BU

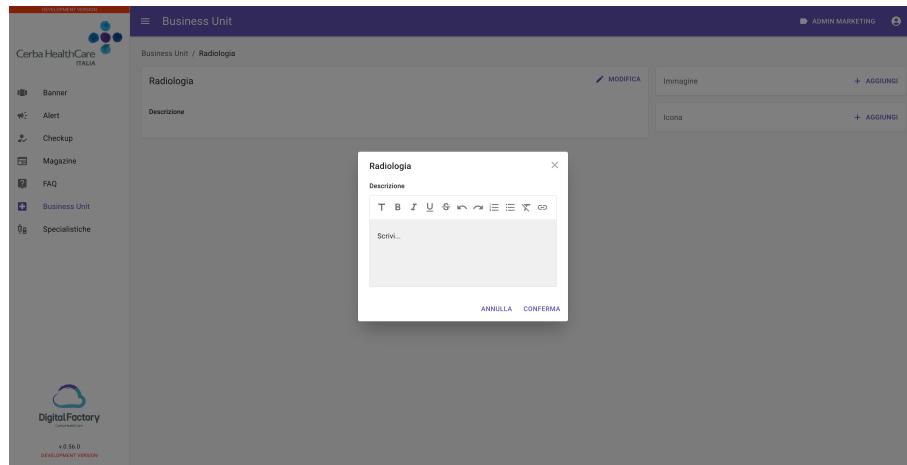


Figura 5.20: Modale modifica descrizione BU

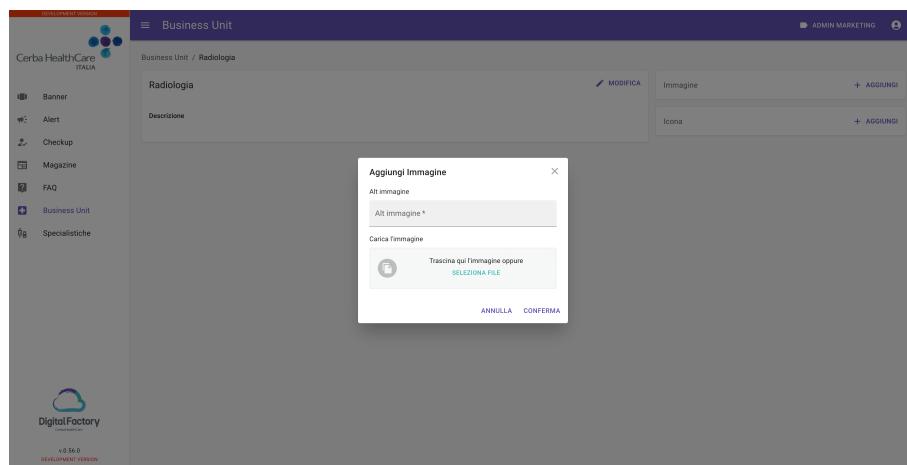


Figura 5.21: Modale aggiunta immagine BU

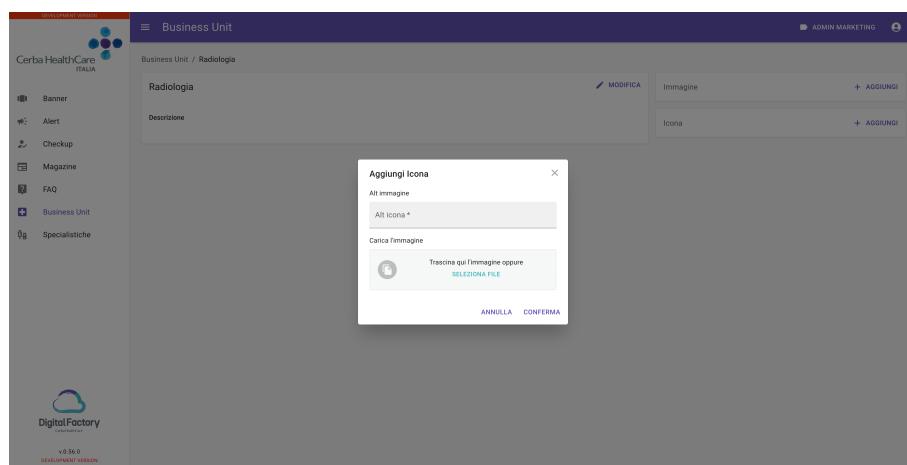


Figura 5.22: Modale aggiunta icona BU

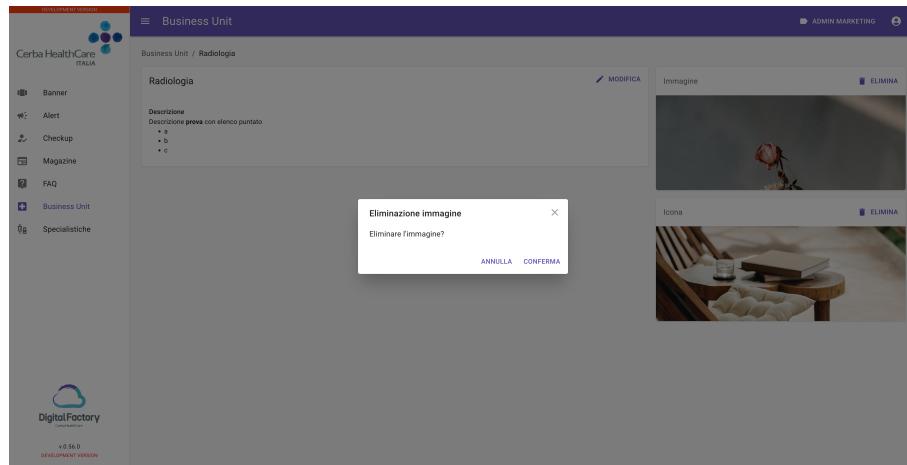


Figura 5.23: Modale eliminazione immagine BU

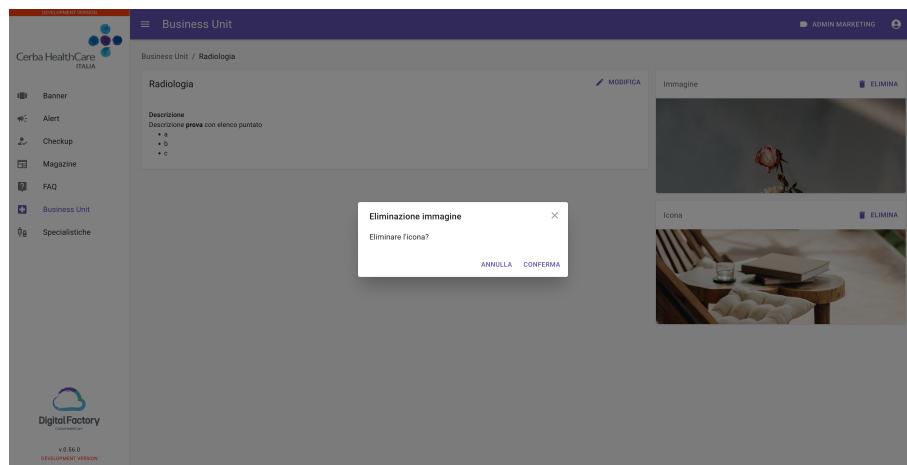


Figura 5.24: Modale eliminazione icona BU

#### 5.5.4 F9: “Specialistiche”

**Requisiti** Aggiungere la sezione “Specialistiche” fra quelle previste per il ruolo utente *marketing admin*.

Al suo interno, una tabella deve mostrare le specialistiche esistenti in una lista semplice e riportarne la proprietà “Nome”.

L’*header* della tabella delle specialistiche deve essere dotato di un campo di ricerca. Questo deve essere in grado di filtrare per nome e posizionato all’estrema destra dell’*header* della tabella. Ogni specialistica deve essere dotata di una pagina di dettaglio alla quale si accede cliccando sul link posto in concomitanza del nome. Qui, deve essere possibile modificare e visualizzare la descrizione, proprietà che deve poter essere formattata; sempre all’interno della pagina di dettaglio, si devono poter aggiungere, visualizzare ed eliminare un’immagine e un’icona.

### 5.5.5 F10: “Posizioni aperte”

**Requisiti** Aggiungere la sezione “Posizioni aperte” fra quelle previste per il ruolo utente *hr admin*.

Al suo interno, una tabella deve mostrare le posizioni aperte esistenti in una lista semplice e riportarne la proprietà “Città”, “Data”, “Area” e “Nome”.

Deve essere possibile aggiungere o eliminare una posizione.

L’*header* della tabella delle posizioni aperte deve essere dotato di un campo di ricerca. Questo deve essere in grado di filtrare per nome e posizionato posizionato alla sinistra del bottone per l’aggiunta di una nuova posizione.

Ogni posizione aperta deve essere dotata di una pagina di dettaglio alla quale si accede cliccando sul link posto in concomitanza del nome. Qui, deve essere possibile modificare e visualizzare i campi delle informazioni, e solamente per la proprietà “Descrizione” deve essere fornita la possibilità di formattazione.

#### Output grafico

Posizioni aperte			
Data	Città	Area	Nome
29/11/2022	Brescia	Area Test	Posizione Prova
28/11/2022	Milano	Area Test	Posizione Esempio
27/11/2022	Torino	Area Test	Posizione Esperimento

Figura 5.25: Tabella posizioni aperte campo ricerca vuoto

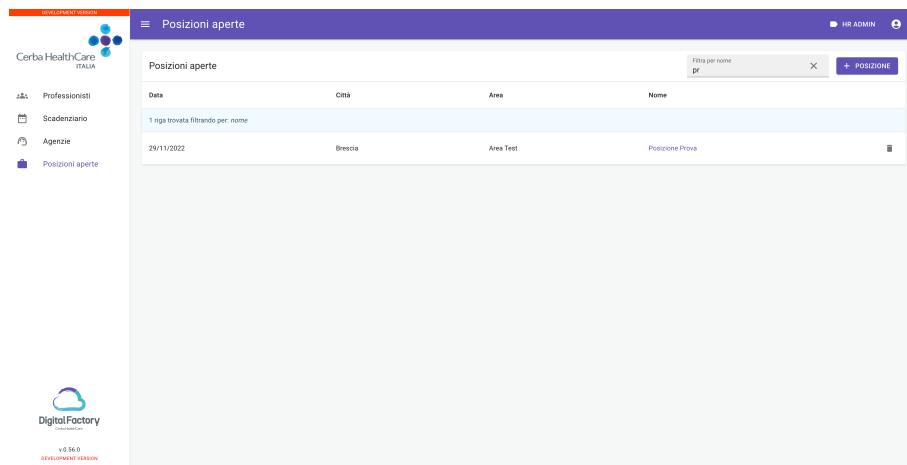


Figura 5.26: Tabella posizioni aperte campo ricerca riempito

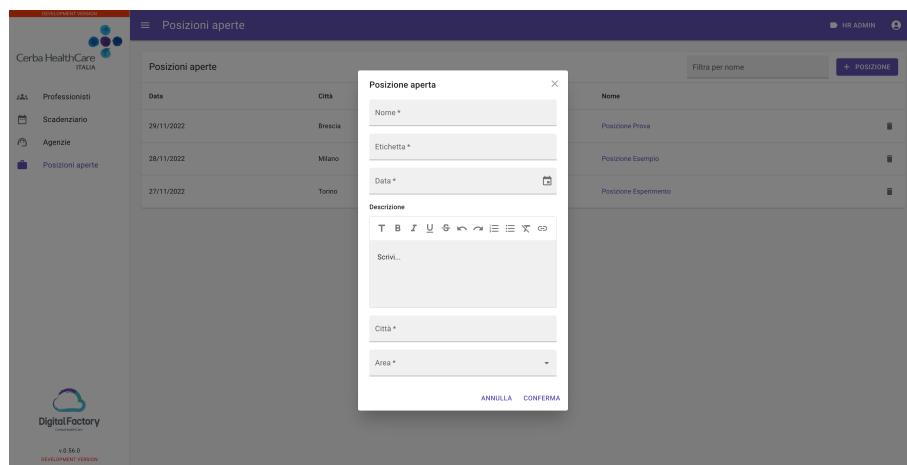


Figura 5.27: Modale aggiunta posizione aperta

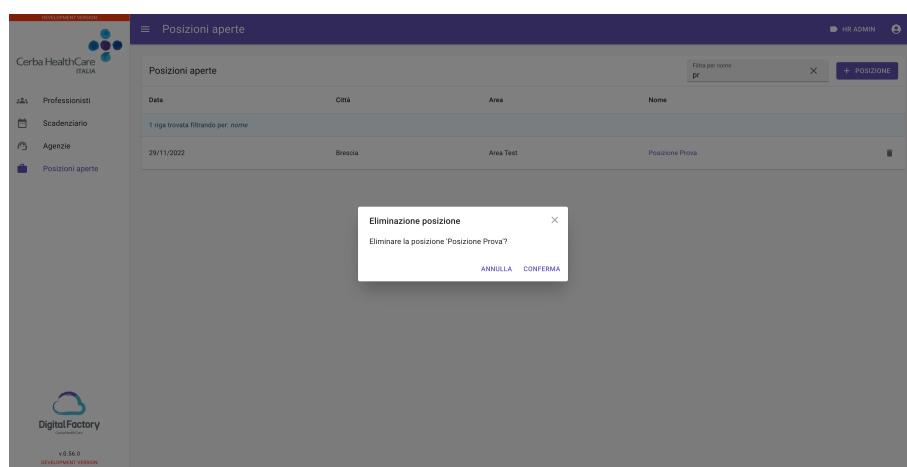


Figura 5.28: Modale eliminazione posizione aperta

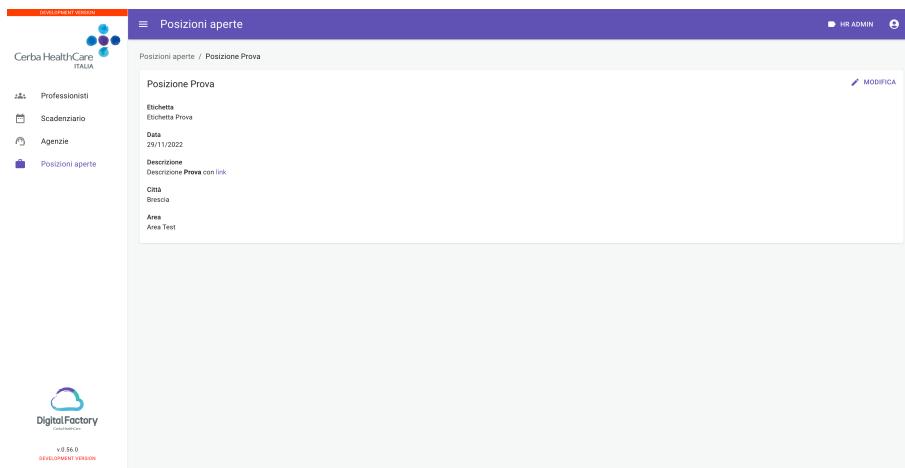


Figura 5.29: Pagina dettaglio posizione aperta

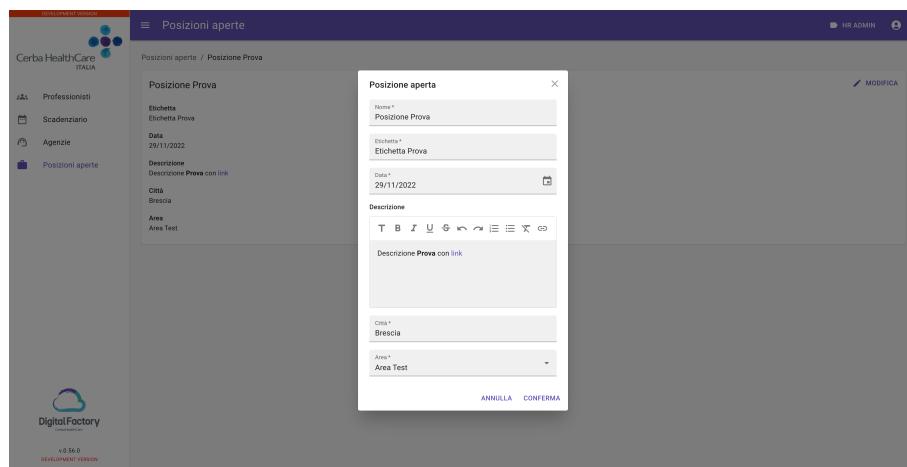


Figura 5.30: Modale modifica posizione aperta



# Conclusioni

L'attività progettuale condotta ai fini dell'elaborato ha rappresentato un'esperienza di grande valore, che mi ha permesso di raggiungere diversi traguardi personali e di crescere professionalmente.

*In primis*, ho potuto relazionarmi per la prima volta con il mondo del lavoro, nello specifico, interfacciandomi con una realtà disponibile e moderna come lo è Link-Up. Grazie all'inserimento in tale contesto aziendale, infatti, ho potuto sperimentare cosa significhi operare come sviluppatore *front-end*, la figura professionale che ambisco a ricoprire in futuro, e far parte di un team interfunzionale in cui si collabora per far fronte a obiettivi comuni.

In secondo luogo, ho avuto l'opportunità di misurarmi con un progetto stimolante, collocato nell'innovativo e impattante contesto della Telemedicina. I vari mesi dedicati allo sviluppo delle implementazioni previste per il modulo “OMNIA”, di fatto, si sono tradotti da un lato in un applicativo aggiornato, che include le evolutive richieste dal committente, e che andrà a incidere positivamente sul flusso lavorativo di coloro che lo utilizzeranno quotidianamente, e dall'altro nell'oggetto protagonista della mia prova finale.

In aggiunta, nel corso dell'attività di sviluppo ho potuto accrescere le conoscenze in mio possesso attraverso i preziosi consigli ricevuti dai professionisti con i quali sono venuto a contatto, e applicare varie nozioni apprese tramite gli insegnamenti del corso di laurea.

Ancora una volta, va sottolineata l'importanza ricoperta delle librerie React e Material UI. Dato il fondamentale ruolo da esse ricoperto all'interno del progetto, con il loro impiego ho avuto l'opportunità di cimentarmi con due tecnologie estremamente al passo con i *trend* attuali e di rilievo per gli sviluppatori *front-end*. Il loro utilizzo ha suscitato in me il desiderio di testare altre soluzioni che orbitano attorno al mondo di JavaScript e di React, nell'ottica di migliorare e accrescere l'attuale bagaglio di conoscenze. Ad ora, fra gli strumenti che più attirano il mio interesse vi sono React Native ed Electron, due *framework* che permettono di sfruttare competenze di sviluppo web per la creazione di applicazioni *mobile*. Ho avuto poi la possibilità di integrare all'interno della mia *routine* lavorativa la metodologia agile di tipo *Scrum*. Ciò mi ha consentito di familiarizzare con uno strumento largamente impiegato nell'ambito dello sviluppo del *software*, e di rapportarmi con pregi e difetti di una strategia altamente dinamica, che, nel mio caso, ha previsto uno stretto contatto con il committente, con altri sviluppatori, e, alle volte, anche con il team di design.

---

Inoltre, ho potuto imbattermi in problemi e situazioni critiche caratteristiche di un’attività di sviluppo, nonché di un progetto così ambizioso. Infatti, è capitato di avere difficoltà nell’ambientarsi e nel destreggiarsi all’interno di un progetto già avviato, gestito da persone dotate di un maggior livello di competenza, nell’utilizzo di soluzioni tecnologiche che, in una certa misura, rappresentano una novità, oppure ancora nel dover adattare l’attività in svolgimento alle esigenze della prova finale.

In conclusione, per quanto concerne il futuro dell’applicativo, il modulo “OMNIA” continuerà a essere gestito con le stesse modalità analizzate nel corso dell’elaborato. Per esso, seppur in misura minore rispetto al passato, sono programmate ulteriori evolutive che porteranno sia all’implementazione di nuove funzionalità per il supporto ad altri team, che alla loro ottimizzazione attraverso l’integrazione del *feedback* ricevuto.

Invece, lo sviluppo della piattaforma per la digitalizzazione dei flussi di lavoro proseguirà perseguiendo due obiettivi principali: per prima cosa, altri moduli che lo compongono, a rotazione (similmente a come è avvenuto per “OMNIA”), diverranno il fulcro dell’attività lavorativa del team per l’aggiunta di nuove funzioni; in secondo luogo, l’applicazione *mobile*, esistente ma ancora in stato embrionale, verrà migliorata per avvicinarsi sempre di più all’essere uno strumento usabile e completo.

# Bibliografia e Sitografia

- [1] Atlassian. *Scrum. Panoramica*. URL:  
<https://www.atlassian.com/it/agile/scrum> [Consultato il 15/11/2022].
- [2] AWS. *Che cos'è un'applicazione web*. URL:  
<https://aws.amazon.com/it/what-is/web-application/> [Consultato il 15/02/2023].
- [3] Axios. *Getting Started. Introduction*. URL: <https://axios-http.com/docs/intro> [Consultato il 28/01/2023].
- [4] R. Barger. *React Context for Beginners – The Complete Guide (2021). What problems does React context solve?* FreeCodeCamp. Lug. 2021. URL:  
<https://www.freecodecamp.org/news/react-context-for-beginners/> [Consultato il 25/01/2023].
- [5] K. Beck e colleghi. *Manifesto per lo Sviluppo Agile di Software*. 2001. URL:  
<https://agilemanifesto.org/iso/it/manifesto.html> [Consultato il 17/11/2022].
- [6] CERN. *A short history of the Web*. URL: <https://home.web.cern.ch/science/computing/birth-web/short-history-web> [Consultato il 14/02/2023].
- [7] D. De Masi. *Smart working. La rivoluzione del lavoro intelligente*. 1<sup>a</sup> ed. Italia: Marsilio, 2020. ISBN: 9788829705696.
- [8] K. Harsh. *Cos'è l'Architettura delle Applicazioni Web? Analisi di un'Applicazione Web*. Kinsta. Mar. 2014. URL: <https://kinsta.com/it/blog/architettura-applicazioni-web/#cos-larchitettura-delle-applicazioni-web> [Consultato il 15/02/2023].
- [9] IBM. *Architettura three-tier*. Ott. 2020. URL:  
<https://www.ibm.com/it-it/cloud/learn/three-tier-architecture> [Consultato il 15/02/2023].

- [10] Istituto Poligrafico e Zecca dello Stato. *LEGGE 22 maggio 2017, n. 81, articolo 18 (capo II), comma 1.* Giu. 2017. URL: <https://www.normattiva.it/uri-res/N2Ls?urn:nir:stato:legge:2017-05-22;81!vig=> [Consultato il 15/11/2022].
- [11] jquense. *Yup.* URL: <https://github.com/jquense/yup#yup> [Consultato il 30/01/2023].
- [12] Meta Platforms Inc. *API di Riferimento degli Hooks. useReducer.* URL: <https://it.reactjs.org/docs/hooks-reference.html#usereducer> [Consultato il 26/01/2023].
- [13] Meta Platforms Inc. *Introduzione a JSX.* URL: <https://it.reactjs.org/docs/introducing-jsx.html> [Consultato il 02/12/2022].
- [14] Meta Platforms Inc. *Introduzione agli Hooks. Le classi confondono sia le persone che le macchine.* URL: <https://it.reactjs.org/docs/hooks-intro.html#classes-confuse-both-people-and-machines> [Consultato il 25/01/2023].
- [15] Meta Platforms Inc. *Introduzione agli Hooks. Retrocompatibili.* URL: <https://it.reactjs.org/docs/hooks-intro.html#no-breaking-changes> [Consultato il 25/01/2023].
- [16] Meta Platforms Inc. *Panoramica sugli Hooks. Regole degli Hooks.* URL: <https://it.reactjs.org/docs/hooks-overview.html#rules-of-hooks> [Consultato il 25/01/2023].
- [17] Meta Platforms Inc. *SyntheticEvent. Supported Events.* URL: <https://reactjs.org/docs/events.html#supported-events> [Consultato il 03/01/2023].
- [18] Meta Platforms Inc. *Usare l'Hook Effect. Example Using Hooks.* URL: <https://it.reactjs.org/docs/hooks-effect.html#example-using-hooks> [Consultato il 25/01/2023].
- [19] Meta Platforms Inc. *Usare l'Hook Effect. Tip: Optimizing Performance by Skipping Effects.* URL: <https://it.reactjs.org/docs/hooks-effect.html#tip-optimizing-performance-by-skipping-effects> [Consultato il 25/01/2023].

- [20] Meta Platforms Inc. *Usare l'Hook State. Cos'è un Hook?* URL:  
<https://it.reactjs.org/docs/hooks-state.html#whats-a-hook> [Consultato il 24/01/2023].
- [21] Meta Platforms Inc. *Usare l'Hook State. Dichiarare una Variabile di Stato.* URL:  
<https://it.reactjs.org/docs/hooks-state.html#declaring-a-state-variable> [Consultato il 24/01/2023].
- [22] K. Mew. *Learning Material Design. Getting Started with Material Design.* 1<sup>a</sup> ed. UK: Packt Publishing Ltd., 2015. ISBN: 9781785289811.
- [23] Ministero della Salute. *TELEMEDICINA, Linee di indirizzo nazionali. Definizione e classificazione dei servizi di Telemedicina.* Mar. 2014. URL:  
[https://www.salute.gov.it/imgs/C\\_17\\_pubblicazioni\\_2129\\_allegato.pdf](https://www.salute.gov.it/imgs/C_17_pubblicazioni_2129_allegato.pdf) [Consultato il 09/02/2023].
- [24] niuware. *mui-rte.* URL: <https://github.com/niuware/mui-rte#mui-rte> [Consultato il 04/02/2023].
- [25] npm. *date-fns.* URL: <https://date-fns.org/> [Consultato il 30/01/2023].
- [26] *React useContext Hook. React Context.* W3C. URL:  
[https://www.w3schools.com/react/react\\_usecontext.asp](https://www.w3schools.com/react/react_usecontext.asp) [Consultato il 25/01/2023].
- [27] Red Hat. *Cos'è la metodologia agile? Panoramica.* Lug. 2022. URL: <https://www.redhat.com/it/devops/what-is-agile-methodology#panoramica> [Consultato il 20/11/2022].
- [28] Red Hat. *Cosa sono le API? Panoramica.* URL:  
<https://www.redhat.com/it/topics/api/what-are-application-programming-interfaces#panoramica> [Consultato il 27/01/2023].
- [29] Red Hat. *I vantaggi delle API. Panoramica.* URL:  
<https://www.redhat.com/it/topics/api#panoramica> [Consultato il 27/01/2023].
- [30] Remix. *React Router.* URL: <https://v5.reactrouter.com/> [Consultato il 06/02/2023].
- [31] W3C. *A Little History of the World Wide Web.* URL:  
<https://www.w3.org/History.html> [Consultato il 14/02/2023].