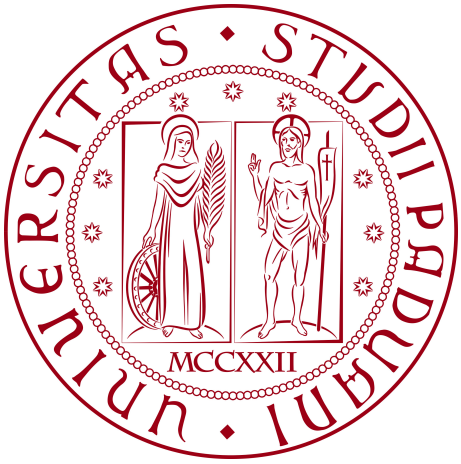
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Sviluppo e gestione di un'app multipiattaforma in monorepo: Caso studio presso UNOX S.p.A.

Tesi di Laurea Triennale

Relatore

Prof. Da San Martino Giovanni

Laureando

Bobirica Andrei Cristian

Matricola 1224449

Anno Accademico 2023-2024

© Bobirica Andrei Cristian , Luglio 2024. Tutti i diritti riservati. Tesi di Laurea Triennale: "*Sviluppo e gestione di un'app multipiattaforma in monorepo: Caso studio presso UNOX S.p.A.*", Università degli Studi di Padova, Dipartimento di Matematica "Tullio Levi-Civita".

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."

— Bjarne Stroustrup.

Ringraziamenti

Desidero esprimere la mia gratitudine al professor Da San Martino Giovanni, mio relatore, per l'aiuto e il sostegno che mi ha dato durante la stesura dell'elaborato.

Un grazie di cuore ai miei genitori, che mi hanno sempre supportato e incoraggiato in ogni fase della mia vita. Senza di loro, nulla di tutto questo sarebbe stato possibile.

Un ringraziamento va alle mie maestre delle elementari, Ornella e Luigina. Quando sono arrivato in Italia all'età di cinque anni, non conoscevo la lingua e mi trovavo di fronte a un nuovo mondo. Loro mi hanno accolto con affetto e pazienza, aiutandomi a integrarmi, a imparare l'italiano e a sentirmi parte di questa nuova realtà. Grazie al loro sostegno e alla loro dedizione, ho potuto costruire le basi per il mio percorso educativo.

Vorrei dedicare un sincero ringraziamento ai miei amici. Grazie di cuore per aver sempre creduto in me, per le innumerevoli ore di studio condivise insieme e per tutto il sostegno che mi avete dato lungo il cammino. Il vostro supporto è stato prezioso e fondamentale per ogni passo del mio percorso. Senza di voi, non avrei potuto raggiungere i traguardi che ho conquistato.

Padova, Luglio 2024 *Bobirica Andrei Cristian*

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di trecentoventi ore, dal laureando Bobirica Andrei Cristian presso l'azienda UNOX S.p.A.
L'obiettivo dello stage era la realizzazione di un'applicazione multipiattaforma che riuscisse a garantire compatibilità con *IOS*, *Android* e *Web*.
La sfida nella realizzazione di questa *app* è stata integrarla con un Design System$_G$ già esistente e in una monorepo$_G$ dove era già presente un'altra applicazione.
In questo documento si potrà esaminare l'analisi tecnica effettuata per l'applicazione, ma anche le problematiche riscontrate nella realizzazione e i spunti di riflessione che ne conseguono.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici sorgenti

Acronimi e abbreviazioni

API Application Programming Interface. 12, 14

DDC Data Driven Cooking. 4–6

E2E End To End. 8, 70

RMA Return Merchandise Authorization. 7

Glossario

Backend La parte dell'applicazione che gestisce la logica di business, il database, e le operazioni di server. Il backend supporta il frontend fornendo dati e funzionalità. 6, 11, 14, 17, 63

CI/CD Integrazione continua e consegna continua. L'integrazione continua (CI) è una pratica di sviluppo software in cui i membri del team integrano il loro lavoro frequentemente, con verifiche automatizzate per rilevare errori rapidamente. La consegna continua (CD) estende la CI, automatizzando ulteriormente il processo di rilascio per consentire la distribuzione di software in produzione in modo rapido e affidabile.. 70

cross-platform Un approccio allo sviluppo software che permette di creare applicazioni compatibili con più sistemi operativi o piattaforme con un unico codice sorgente. 1

DDC Una piattaforma di cucina intelligente che utilizza i dati per ottimizzare e migliorare i processi di cottura. DDC fornisce funzionalità avanzate per i proprietari di forni, consentendo loro di monitorare e controllare i dispositivi in modo efficiente. 68

DDC Service Un'applicazione destinata al personale tecnico e di manutenzione dei forni, offrendo strumenti avanzati per la gestione e il supporto dei dispositivi. DDC Service facilita la risoluzione dei problemi e l'ottimizzazione delle operazioni di servizio. 5, 11, 13, 14, 17, 26, 30, 68

Design Pattern Soluzioni riutilizzabili a problemi comuni di progettazione nel software. I design pattern facilitano la creazione di codice robusto e manutenibile. 2

Design System Un insieme di linee guida, componenti riutilizzabili e strumenti per creare un'interfaccia utente coerente e unificata. Il Design System garantisce uniformità visiva e comportamentale in tutte le parti dell'applicazione. iv, 5, 6, 9, 25

DevOps Una pratica che combina lo sviluppo software (Development) e le operazioni IT (Operations) per migliorare la collaborazione e la produttività, automatizzando i processi di sviluppo, test e distribuzione del software. 11

ECN Per ECN si intende un codice che identifica il versionamente di uno specifico forno, un diverso versionamento potrebbe indicare caratteristiche diverse di un medesimo prodotto. 23

Firmware Il software integrato nei dispositivi hardware, come i forni, che gestisce le loro operazioni fondamentali. Gli aggiornamenti del firmware migliorano le prestazioni e aggiungono nuove funzionalità ai dispositivi. 24

Frontend La parte dell'applicazione con cui gli utenti interagiscono direttamente. Comprende l'interfaccia utente e la logica di presentazione. 11, 14, 63

JWT Un formato compatto e sicuro per la trasmissione di informazioni tra parti come un oggetto JSON. I JWT sono spesso utilizzati per l'autenticazione e l'autorizzazione nelle applicazioni web. 41

monorepo Una strategia di gestione del codice sorgente in cui più progetti vengono memorizzati in un unico repository. Il monorepo facilita la condivisione del codice e la gestione delle dipendenze tra i progetti. iv, 1, 2, 5, 6, 13, 30, 63

Nav Bar La barra di navigazione dell'applicazione che consente agli utenti di accedere rapidamente alle diverse sezioni e funzionalità. 24

Product Code Un identificativo univoco assegnato a ogni prodotto per la tracciabilità e la gestione. Utilizzato nelle applicazioni per monitorare e gestire specifici modelli di forni. 23

Repo Un archivio centralizzato dove il codice sorgente e altri file di un progetto vengono memorizzati e gestiti, spesso utilizzato con sistemi di controllo versione come Git. 13, 63

RTK Un toolkit per Redux che semplifica la scrittura della logica Redux e automatizza configurazioni complesse. RTK è stato utilizzato per gestire lo stato globale dell'applicazione in modo efficiente e strutturato. 38

Serviced Oven Nel contesto della applicazione DDC Service per Serviced Oven si intente un prodotto, in particolare un forno. Questo forno è un prodotto a cui un addetto Service presta servizzi di assistenza e riparazione. Un Utente Autenticato della app DDC Service ha tra i sui Serviced Ovens i prodotti a cui presta assistenza e servizi di Service. Un Serviced Oven è un prodotto fisico realmente esistente edentificato da un seriale.. 20, 24

Tab Bar Una barra di navigazione a schede che permette agli utenti di passare facilmente tra diverse schermate o funzionalità dell'applicazione. 24

Capitolo 1

Introduzione

1.1 Convenzioni tipografiche

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- Gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento.
- Per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*$_{G}$.
- I termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

1.2 Organizzazione del testo

Questa tesi è strutturata per fornire una visione dettagliata e comprensibile dell'esperienza di stage presso UNOX S.p.A., focalizzandosi sullo sviluppo di un'applicazione cross-platform$_{G}$ e sullo studio di un ambiente di sviluppo in monorepo$_{G}$. La suddivisione dei capitoli permette di seguire il percorso progettuale in modo chiaro e logico, dal contesto aziendale alle conclusioni finali. Di seguito è riportata l'organizzazione del testo:

Introduzione: Il capitolo introduttivo presenta una panoramica dell'azienda UNOX S.p.A., il contesto dello stage, e fornisce una descrizione dettagliata dell'organizzazione della tesi.

Descrizione dello stage: In questo capitolo viene descritto il progetto di stage, inclusi gli obiettivi tecnici e professionali, le attività svolte, la pianificazione dettagliata e l'analisi preventiva dei rischi.

Tecnologie utilizzate: Questo capitolo elenca e descrive le tecnologie impiegate durante lo sviluppo dell'applicazione.

Analisi dei requisiti: In questa sezione vengono descritti i casi d'uso, il monitoraggio dei requisiti e le tabelle che specificano le funzioni principali dell'applicazione.

Progettazione e codifica: In questo capitolo verranno esaminati i Design Pattern$_{G}$ adottati, esemplificati con parti significative di codice, insieme a descrizioni dettagliate di alcune funzionalità chiave sviluppate.

Studio fattibilità app in monorepo: Questo capitolo esplora la fattibilità dello sviluppo dell'applicazione in un ambiente monorepo$_{G}$, descrivendo l'organizzazione iniziale, le problematiche rilevate, le soluzioni proposte e gli strumenti utilizzati per la gestione delle dipendenze.

Conclusioni: Il capitolo conclusivo presenta un consuntivo finale del lavoro svolto, una valutazione del raggiungimento degli obiettivi prefissati, le conoscenze acquisite durante lo stage, e una riflessione personale sull'esperienza complessiva.

Bibliografia e Sitografia: Infine, vengono elencate le fonti bibliografiche e sitografiche consultate per la redazione della tesi.

1.3 L'azienda

UNOX S.p.A.è un'azienda leader nel settore della produzione di forni professionali per la ristorazione, fondata nel 1990 e situata a Cadoneghe, in provincia di Padova, Italia.
Riconosciuta a livello internazionale per la qualità, l'affidabilità e l'innovazione dei suoi prodotti, UNOX è all'avanguardia nella tecnologia di cottura intelligente, che integra connettività avanzata e automazione.[1]

1.3.1 Mission e Vision

La mission di UNOX S.p.A.è quella di contribuire al successo dei propri clienti offrendo soluzioni innovative e di alta qualità che migliorano le prestazioni e l'efficienza delle loro cucine. L'azienda si impegna a fornire prodotti che combinano tecnologia avanzata e facilità d'uso, garantendo al contempo sostenibilità ambientale e risparmio energetico.
La vision di UNOX si concentra sull'essere il punto di riferimento per l'innovazione nel settore della ristorazione professionale. L'azienda punta a creare valore attraverso lo sviluppo continuo di tecnologie all'avanguardia e il miglioramento costante dei propri prodotti e servizi.

1.3.2 Prodotti e Servizi

UNOX offre una vasta gamma di forni professionali, noti per la loro efficienza, versatilità e innovazione tecnologica. I prodotti principali includono:

- Forni a convezione: Forni che utilizzano l'aria calda per cuocere il cibo in modo uniforme e veloce.
- Forni a vapore: Forni che utilizzano il vapore per cucinare in modo sano e preservare le proprietà nutrizionali degli alimenti.
- Sistemi di cottura intelligenti: Tecnologie integrate che permettono il controllo preciso dei processi di cottura e l'automazione delle operazioni.

[1] *Unox*. URL: https://www.unox.com/it_it/.

- Forni combinati: Forni che combinano cottura a vapore e a convezione, ideali per una varietà di preparazioni culinarie.

1.3.3 Connettività e Innovazione

UNOX S.p.A.è pioniera nell'integrazione della connettività nei suoi prodotti, offrendo soluzioni che permettono il monitoraggio e il controllo remoto dei forni attraverso piattaforme digitali.
L'azienda ha sviluppato il progetto Data Driven Cooking (DDC)$_G$, una piattaforma che utilizza i dati raccolti dai forni per ottimizzare i processi di cottura e fornire suggerimenti personalizzati agli chef.

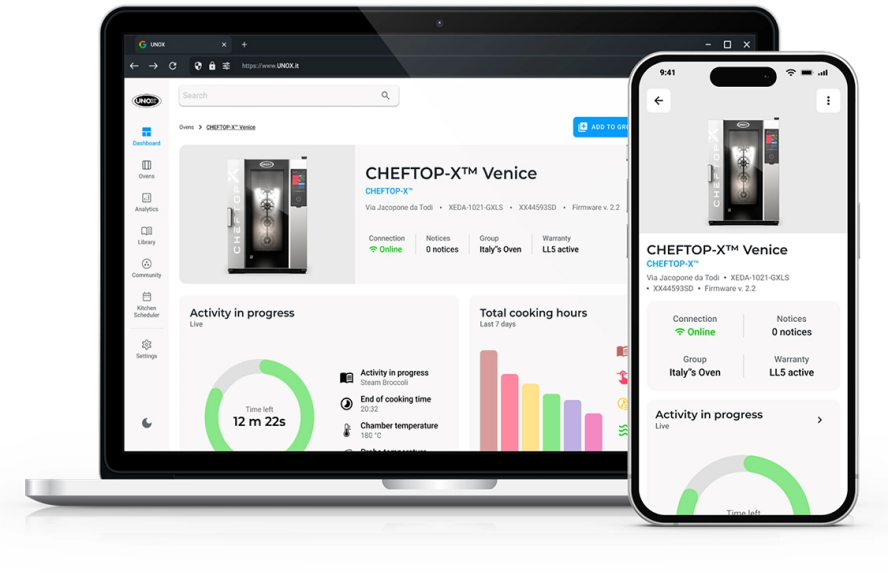


Figura 1.1: App *DDC*: Applicazione DDC su piattaforma Web e Mobile

1.4 Lo stage

L'offerta di stage mi ha immediatamente intrigato per il suo focus su un progetto specifico vitale per l'azienda, piuttosto che un semplice esercizio accademico. Questo aspetto ha richiesto un impegno significativo e una collaborazione intensa con diversi

team per portare a termine un progetto cruciale per l'azienda. Nel contesto aziendale esistono due app chiamate DDC$_G$ e DDC Service$_G$.

- *DDC* ha come utilizzatori i proprietari dei forni che utilizzano questa app per le funzionalità connesse dei loro dispositivi.
- *DDC Service* ha come utilizzatori personale tecnico, personale responsabile di manutenzione dei forni e utenti addetti al *Service*.

Per rispondere alle esigenze aziendali, è stato necessario avviare lo sviluppo di una nuova app *DDC Service* con una prospettiva moderna e che sia multi-piattaforma. Durante il mio stage presso UNOX S.p.A., ho lavorato principalmente sull'avvio dello sviluppo di questa app, il mio obiettivo principale è stato ristrutturare e sviluppare completamente da zero *DDC Service* precedentemente limitata alla piattaforma *Web*. Ho esteso le funzionalità di *DDC Service* per renderla compatibile con dispositivi *Android, iOS e Web*, integrando questa nuova versione nell'esistente monorepo$_G$ di *DDC*.
Questo approccio ha permesso di condividere il Design System$_G$ e sfruttare l'infrastruttura esistente per ottimizzare l'efficienza e la manutenibilità del codice.
Durante il periodo di stage, ho collaborato attivamente con il team di sviluppo, design e progettazione per implementare le prime funzionalità richieste per l'applicazione, rispettando le linee guida e assicurando la compatibilità su tutte le piattaforme *target*. Questa esperienza mi ha fornito competenze pratiche nello sviluppo software multi-piattaforma e una comprensione approfondita della progettazione scalabile e della gestione delle risorse tecniche in un ambiente *monorepo*.

Capitolo 2

Descrizione dello stage

2.1 Pianificazione

2.1.1 Attività

La seguente pianificazione delle attività è stata inizialmente delineata nel piano di lavoro. Tuttavia, durante lo stage, alcune attività sono state modificate sia per una maggiore comprensione emersa dall'analisi dei requisiti, sia per cambiamenti negli obiettivi da realizzare.

Prima Settimana (40 ore)

- Incontro con le persone coinvolte nel progetto per discutere i requisiti e le richieste relative al sistema da sviluppare.
- Verifica delle credenziali e degli strumenti di lavoro assegnati.
- Presa visione dell'infrastruttura esistente, in particolare della app DDC$_{G}$, del suo Design System$_{G}$ e della monorepo$_{G}$ esistente.
- Formazione sulle tecnologie adottate.

Seconda Settimana (40 ore)

- Studio del software Backend$_{G}$ esistente con cui l'applicazione si integrerà.

- Avvio dello sviluppo dell'applicazione, definizione dell'architettura, dello *stack* di navigazione e implementazione della funzionalità di autenticazione.

Terza Settimana (40 ore)

- Continuazione dello sviluppo dell'architettura dell'applicazione, inclusi il *login* e lo *stack* di navigazione principale.

Quarta Settimana (40 ore)

- Sviluppo della funzionalità consultazione Prodotto, detta *Product Page*

Quinta Settimana (40 ore)

- Continuazione dello sviluppo delle funzionalità di consultazione Prodotto.

Sesta Settimana (40 ore)

- Implementazione nella *Product Page* delle funzionalità di visualizzazione manuali, ricambistica e *Tech and Docs*
- Sviluppa della funzionalità consultazione *Serviced Oven*

Settima Settimana (40 ore)

- Sviluppo della funzionalità di gestione del flusso Return Merchandise Authorization (RMA)$_G$.

Ottava Settimana - Conclusione (40 ore)

- Continuazione dello sviluppo della funzionalità di gestione del flusso *RMA*.
- Test e ottimizzazione dell'applicazione con il personale aziendale.

Durata in ore	Descrizione dell'attività
40	Inserimento in azienda
24	Studio *Backend* esistente
56	Sviluppo architettura applicazione
80	Sviluppo della funzionalità *Product Page*
40	Sviluppo della funzionalità *Serviced Oven Page*
60	Sviluppo funzionalità flusso *RMA*
20	Test e ottimizzazione della applicazione
Totale ore	**320**

Tabella 2.1: Suddivisione delle ore di lavoro per le attività di progetto.

2.1.2 Obbiettivi

2.1.2.1 Obiettivi obbligatori

- **Architettura dell'applicazione:** Definizione dello scheletro e dell'architettura dell'applicazione, compresa la navigazione.
- **Autenticazione:** Implementazione della funzionalità di autenticazione (*SignIn, SignUp, Recover Password*).
- ***Product Page*:** Sviluppo della funzionalità consultazione Prodotto e delle funzionalità di visualizzazione manuali, ricambistica e *Tech and Docs*.
- ***Serviced Oven*:** Sviluppa della funzionalità consultazione dei propri forni in *service* detti *Serviced Oven*.
- **Test piattaforme:** Esecuzione di test sulla piattaforma *web* e *mobile* per garantire la massima portabilità del codice.

2.1.2.2 Obiettivi desiderabili

- **RMA:** Sviluppo della funzionalità di gestione del flusso *RMA*.

2.1.2.3 Obiettivi facoltativi

- **Test E2E:** Creazione di test automatizzati End To End (E2E)$_G$ per verificare le varie componenti dell'applicazione, per massimizzare l'efficienza del processo di *testing*.

2.1.3 Vincoli

Durante lo sviluppo del progetto, sono stati identificati vari vincoli che hanno influenzato il contesto operativo e le decisioni progettuali. Questi vincoli hanno avuto un impatto significativo sulle scelte effettuate e sull'approccio adottato per la realizzazione dell'applicazione. I principali vincoli sono suddivisibili in categorie come vincoli aziendali, tecnologici, temporali e di design.

2.1.3.1 Vincoli tecnologici

Un vincolo importante riguardava l'adozione delle tecnologie già utilizzate da UNOX S.p.A.L'applicazione doveva essere sviluppata utilizzando strumenti e tecnologie in uso all'interno dell'azienda per assicurare l'integrazione e la coerenza con l'ecosistema tecnologico esistente.

2.1.3.2 Vincoli temporali

Un vincolo temporale significativo era la data di conclusione dello stage, fissata per il 7 giugno 2024. Questa scadenza ha imposto un termine rigido per il completamento del progetto, richiedendo una gestione attenta del tempo e delle risorse per rispettare il limite prestabilito.

2.1.3.3 Vincoli di *design*

I vincoli di design includevano il rispetto del Design System$_G$ aziendale e delle specifiche grafiche fornite dall'azienda. L'applicazione doveva essere allineata al *Design System* esistente e rispettare le palette di colori e le linee guida visive stabilite dall'azienda.

2.2 Analisi preventiva dei rischi

Durante la fase di analisi iniziale, sono stati individuati alcuni possibili rischi che avrebbero potuto causare problemi nel corso del progetto. Per affrontarli, sono state elaborate delle possibili soluzioni.

1. Inesperienza tecnologica

Descrizione: Era previsto l'utilizzo di tecnologie mai utilizzate prima, il che poteva causare rallentamenti nello sviluppo dell'applicazione.

Soluzione: L'azienda ha programmato un periodo di circa una settimana dedicato allo studio autonomo delle tecnologie, utilizzando tutorial e risorse interne.

2. Difficoltà nel soddisfare le esigenze di *design*

Descrizione: Inizialmente era previsto realizzare le funzionalità richieste senza dare peso al *design* e alla parte grafica dell'app. Tuttavia, è emersa la necessità di cooperare con il team di *design* per seguire le loro linee guida.

Soluzione: Si è utilizzato il *Design System* già esistente, adattandolo dove necessario, e si è dato del tempo per imparare a utilizzare nuovi strumenti come Figma.

3. Interpretazione dei requisiti

Descrizione: I requisiti avrebbero potuto subire aggiornamenti in corso d'opera a causa della difficoltà d'individuare con facilità se un requisito fosse realizzabile o meno.

Soluzione: Sono stati pianificati *meeting* regolari con i *team* coinvolti per discutere e individuare soluzioni, semplificando la realizzazione dei requisiti proposti.

Capitolo 3

Tecnologie utilizzate

Questo capitolo esplora le tecnologie chiave adottate nel contesto dello sviluppo dell'applicazione DDC Service$_{G}$. Vengono presentati i linguaggi di programmazione, i *framework*, gli strumenti di sviluppo, le piattaforme di collaborazione e gestione, oltre alle soluzioni *cloud* e Development and Operations$_{G}$ impiegate per supportare e ottimizzare il processo di sviluppo. Ogni tecnologia è discussa nel contesto del suo ruolo nell'ecosistema di sviluppo dell'applicazione, evidenziando come contribuisca alla scalabilità, alla manutenibilità e alla coerenza del codice, nonché al miglioramento complessivo dell'efficienza operativa dell'azienda.

3.1 Linguaggi di programmazione

- **TypeScript**: è un linguaggio di programmazione *open-source* sviluppato da Microsoft. TypeScript è un *superset* di *JavaScript* che aggiunge la tipizzazione statica opzionale e altre funzionalità moderne, rendendo il codice più robusto e manutenibile. Questo linguaggio è stato utilizzato per lo sviluppo dell'applicazione *DDC Service*, sia per la parte Frontend$_{G}$ che per l'integrazione con i servizi Backend$_{G}$, garantendo una maggiore affidabilità e scalabilità del codice.

3.2 *Framework* in uso

- **Expo**: è un *framework open-source* per la creazione di applicazioni *React Native*. Facilita lo sviluppo di applicazioni *mobile* fornendo strumenti e librerie preconfigurate. *Expo* è stato utilizzato per lo sviluppo delle applicazioni *Android* e *IOS*, permettendo di scrivere il codice una sola volta e distribuirlo su entrambe le piattaforme in modo efficiente.

- **Next.js**: un *framework* di sviluppo *React* per la creazione di applicazioni *Web*. Supporta il *rendering* lato *server* e la generazione di siti statici, migliorando così le prestazioni e l'ottimizzazione per i motori di ricerca (*SEO*).

- **React**: è una libreria *JavaScript* per la costruzione di interfacce utente sviluppata da *Facebook*. React si distingue per la sua architettura basata su componenti e l'uso del *virtual DOM*, che rendono lo sviluppo di interfacce utente reattive ed efficienti. In particolare, React è stato utilizzato come base per le applicazioni, consentendo la creazione di componenti riutilizzabili che migliorano la coerenza e la manutenibilità del codice.

- **React Native**: è un framework open-source per lo sviluppo di applicazioni mobili creato da *Facebook*. React Native permette di utilizzare React e *TypeScript* per costruire applicazioni native per *IOS* e *Android*. Unox utilizza React Native per sviluppare applicazioni mobili, permettendo al team di scrivere il codice una sola volta e distribuirlo su entrambe le piattaforme, semplificando e ottimizzando gli sforzi di sviluppo.

- **NodeJS**: è una piattaforma di *runtime open-source* basata su *JavaScript V8* di *Chrome*, progettata per costruire applicazioni di rete veloci e scalabili. Unox utilizza NodeJS per l'esecuzione del codice *TypeScript* e come gestore di pacchetti. Facilita le operazioni lato *server*, consentendo una gestione efficiente di compiti come il *rendering* del server e lo sviluppo di Application Programming Interface (API)$_G$, migliorando le prestazioni e la scalabilità dell'applicazione.

3.3 Tecnologie per *monorepo*

- **NPM Workspaces**: una funzionalità di NPM che consente di gestire più pacchetti all'interno di un unico Repository$_G$.
  NPM Workspaces è stato utilizzato per organizzare i vari pacchetti del progetto DDC Service$_G$, semplificando la gestione delle dipendenze e migliorando l'efficienza dello sviluppo.

- **NX**: un set di strumenti per la gestione di monorepo$_G$ che facilita lo sviluppo, il test e la manutenzione di applicazioni e librerie su larga scala. Nel progetto *DDC Service*, NX è stato implementato nella parte finale per la gestione del *monorepo*, per organizzare e gestire le dipendenze del codice e migliorando la modularità e la coerenza del progetto.

3.4 Librerie utilizzate

- **Solito**: una libreria che permette di condividere il codice tra applicazioni *Next* ed *Expo*, riducendo la duplicazione del codice e semplificando la manutenzione. Nel contesto del progetto DDC Service$_G$, Solito è stato impiegato per ottimizzare la condivisione del codice tra le piattaforme web e mobile, implementando una gestione del routing comune tra le pagine. Ciò ha permesso di mantenere una struttura di navigazione coerente e una logica di gestione dei percorsi uniforme, migliorando l'esperienza dell'utente e semplificando lo sviluppo e la manutenzione dell'applicazione su entrambe le piattaforme.[1]

- **Moti**: una libreria di animazioni per *React Native*$_G$ che facilita la creazione di animazioni complesse e fluide. È stata utilizzata nel progetto *DDC Service* per migliorare l'interazione dell'utente e l'aspetto visivo delle applicazioni mobili.[2]

- **Dripsy**: una libreria per la gestione dello stile di *UI* per *React Native* e *Web*. Dripsy permette di definire uno stile una sola volta e applicarlo ovunque,

[1] *Solito*. URL: https://solito.dev.

[2] *Moti*. URL: https://moti.fyi/starter.

supportando la creazione di interfacce responsive che si adattano automaticamente a diverse dimensioni di schermo. È compatibile con Expo, Vanilla React Native e Next.js, offrendo un supporto completo per TypeScript e facilitando l'implementazione di temi personalizzati e varianti di tema. Con una semplice API$_G$, è possibile definire stili tematici e responsivi in una sola riga di codice. Supporta anche modalità scura e personalizzazione dei colori.[3]

- **Redux**: una libreria per la gestione dello stato delle applicazioni *JavaScript*. *Redux* è utilizzato per mantenere uno stato globale consistente e prevedibile nell'applicazione, facilitando la gestione dello stato complesso e la sincronizzazione dei dati tra i vari componenti.

- **Redux Toolkit (RTK)**: una serie di strumenti e convenzioni per semplificare l'uso di *Redux*. *RTK* include funzioni per la creazione di *slice* di stato, *middleware* personalizzati e la gestione di operazioni asincrone, rendendo lo sviluppo con *Redux* più efficiente e meno soggetto a errori.

- **GraphQL**: un linguaggio di query per *API*$_G$. *GraphQL* è utilizzato per ottenere dati in modo efficiente e flessibile, consentendo di specificare esattamente quali dati sono necessari. Nel progetto DDC Service$_G$, *GraphQL* facilita la comunicazione tra il Frontend$_G$ e il Backend$_G$, migliorando le performance e riducendo la quantità di dati trasferiti.

- **GraphQL Code Generator**: uno strumento per generare tipi *TypeScript* per le query, le mutazioni e i frammenti definiti nello schema *GraphQL*. Questo migliora la sicurezza del tipo e riduce gli errori di *runtime*, mantenendo il codice sincronizzato con lo schema *GraphQL*.

3.5 Strumenti di sviluppo

- **Visual Studio Code**: un *editor* di codice sorgente sviluppato da *Microsoft*, altamente estensibile e utilizzato per una varietà di linguaggi di programmazione.

[3] *Dripsy*. URL: https://www.dripsy.xyz.

- **Xcode**: un ambiente di sviluppo integrato (*IDE*) di *Apple* per *macOS*, utilizzato per sviluppare software per *IOS*, *macOS*, *watchOS* e *tvOS*.
- **Android Studio**: un *IDE* ufficiale per lo sviluppo di applicazioni *Android*, fornito da *Google*. Viene utilizzato per scrivere, eseguire il debug e testare le applicazioni *Android*.
- **Prettier**: uno strumento di formattazione del codice che aiuta a mantenere uno stile di codice coerente in tutti i progetti.
- **Cocoapods**: un gestore di dipendenze per *Swift* e *Objective-C Cocoa projects*. Viene utilizzato per integrare librerie di terze parti nei progetti *IOS*.

3.6 Piattaforme di collaborazione e gestione

- **Git**: viene utilizzato come sistema di controllo delle versioni per il tracciamento delle modifiche al codice sorgente.
- **Microsoft Teams**: adottato come strumento di comunicazione e collaborazione in tempo reale all'interno dell'azienda, facilitando le discussioni, le videochiamate e la condivisione di documenti. Viene utilizzato anche per la calendarizzazione di eventi e meeting.

3.7 Piattaforme *cloud* e *DevOps*

- **Microsoft Azure**: una piattaforma cloud utilizzata per l'hosting di applicazioni, servizi e dati aziendali. Viene utilizzato da Unox per l'hosting di alcuni dei servizi principali. Dalla suite di Azure, viene utilizzato anche *Azure DevOps* per la gestione delle attività di sviluppo software, tra cui la gestione dei repository Git, delle build e delle attività.
- **AWS**: *Amazon Web Services* (*AWS*) è un altro servizio cloud utilizzato per le risorse di calcolo, archiviazione e servizi di rete. Alcuni dei servizi secondari di Unox sono ospitati su AWS.

- **Amplify**: è una piattaforma di sviluppo di applicazioni cloud che facilita l'integrazione di funzionalità come autenticazione, *API*, *storage* e altro ancora. In questo progetto, *Amplify* è stato utilizzato per scaricare automaticamente le chiavi di accesso, migliorando la sicurezza e semplificando la gestione delle credenziali. Inoltre, Amplify è stato impiegato per implementare le notifiche *push*, permettendo una comunicazione efficace e tempestiva con gli utenti dell'applicazione.

3.8 Strumenti di *design*

- **Figma**: uno strumento di design collaborativo utilizzato per la progettazione delle interfacce utente. Facilita la collaborazione tra *designer* e sviluppatori e permette di creare e condividere facilmente prototipi e *design*.

Capitolo 4

Analisi dei requisiti

4.1 Caratteristiche degli utenti

Per esaminare la *user base* dell'app DDC Service$_G$ bisogna tenere in considerazione che tutti i dati di cui l'applicazione fa uso saranno reperibili dal Backend$_G$ associato all'applicazione. Le basi di dati con cui il *backend* andrà a comunicare saranno popolate da piattaforme esterne ignote alla app in questione. Il sistema di autenticazione non dovrà tenere in considerazione di tipologie di utenti speciali come *Admin* o *SysAdmin*.

Questi utenti possono far parte del personale Unox oppure possono essere professionisti con cui Unox collabora. Sono persone che richiedono di accedere alla app sia da *Desktop* tramite *web-app* mentre sono in ufficio, che da *app* sul proprio *smartphone* mentre sono in mobilità. La user base della app è composta da personale tecnico, personale responsabile alla vendita e utenti addetti al *Service*. Non esiste però la necessità lato autenticazione di rendere il processo di registrazione o di *login* diverso in base alla funzione del utente o in base alla sua appartenenza. In linea generica tutti gli utenti registrati hanno gli stessi permessi. In futuro l'applicazione necessiterà di un sistema per rendere un utente registrato verificato o meno, però per questo progetto questa richiesta non è una necessità.

4.2 Vincoli generali

Date le premesse sulle caratteristiche degli utenti della applicazione, definiremo quindi solo due categorie di utenti: Utente Autenticato, Utente Non Autenticato. Ho individuato i seguenti attori:

Utente Non Autenticato L'Utente Non Autenticato è una tipologia di utente che o non si è ancora registrato e quindi non possiede delle credenziali oppure possiede delle credenziali ma deve ancora far il processo di *login* detto *signin*.

Utente Autenticato L'Utente Autenticato è una tipologia di utente che possiede delle credenziali valide e che ha completato con successo il processo di *login* detto *signin*.

4.3 Casi d'uso

I seguenti casi d'uso sono state formulati solo per le funzionalità che sono state realmente realizzate in quanto le funzionalità opzionali per le quali non si ha avuto tempo non sono state tenute in considerazione per l'analisi dei requisiti.

UC0: Autenticazione

Attori Principali: Utente Non Autenticato
Precondizioni: Un Utente Non Autenticato vuole accedere alle funzionalità della *app*
Descrizione: Questo scenario descrive un utente che vuole accedere alla funzionalità della *app* ma che non ne ha i permessi, per tanto deve fare la registrazione (*singup*) o il *login* (*singin*)
Postcondizioni: l'utente accede al processo di registrazione o di *login*

UC0.1: Registrazione

Attori Principali: Utente Non Autenticato
Precondizioni: Un Utente Non Autenticato NON possiede delle credenziali per

poter accedere alla *app*.

Descrizione: Questo scenario descrive un utente che non possiede credenziali di autenticazione e che non riesce ad accedere alle funzionalità della *app*, per tanto intraprende il processo di registrazione con cui alla fine riuscirà ad avere delle credenziali valide.

Postcondizioni: L'utente ora ha delle credenziali valide con cui poter effettuare il *login* e diventare un Utente Autenticato

Scenario Alternativo: Se l'utente non fornisce con correttezza tutti i dati necessari alla registrazione, visualizza un messaggio di errore.

UC0.2: Login

Attori Principali: Utente Non Autenticato

Precondizioni: Un Utente non Autenticato che possiede delle credenziali valide per l'autenticazione.

Descrizione: Tramite questo scenario l'utente fornisce le sue credenziali per l'autenticazione e completa il processo di *login*.

Postcondizioni: L'utente è diventato un Utente Autenticato

Scenario Alternativo: Se l'utente fornisce delle credenziali sbagliate, visualizza un messaggio di errore.

UC0.3: Recover Password

Attori Principali: Utente Non Autenticato

Precondizioni: Un Utente non Autenticato che possiede delle credenziali NON valide per l'autenticazione.

Descrizione: Tramite questo scenario l'utente fornisce le sue credenziali in maniera parziale e tramite un processo riesce a ripristinarle per poter avere delle credenziali valide.

Postcondizioni: L'utente ha delle credenziali valide.

UC1: Visualizzazione Prodotto

Attori Principali: Utente Autenticato

Precondizioni: Un utente ha completato con successo il processo di autenticazione.

Descrizione: L'utente visualizza le informazioni di un determinato prodotto incluse le sue parti commerciali e i dettagli tecnici a lui associati.

Postcondizioni: L'utente ha visualizzato le informazioni collegate al prodotto richiesto.

Scenario Alternativo: Se il prodotto richiesto non è esistente, viene visualizzato un messaggio di errore.

UC2: Visualizzazione Serviced Oven

Attori Principali: Utente Autenticato

Precondizioni: Un utente ha completato con successo il processo di autenticazione e ha tra i suoi forni alcuni dispositivi categorizzati come *Serviced Oven*, cioè forni a cui l'utente presta assistenza o manutenzione.

Descrizione: L'utente visualizza le informazioni di un determinato Serviced Oven$_{G}$ di cui ne ha i permessi.

Postcondizioni: L'utente ha visualizzato le informazioni collegate al *Serviced Oven* richiesto.

Scenario Alternativo: Se il prodotto richiesto non è esistente, viene visualizzato un messaggio di errore.

4.4 Tracciamento dei requisiti

Da un'attenta analisi dei requisiti e degli use case effettuata sul progetto è stata stilata la tabella che traccia i requisiti in rapporto agli use case.
Sono stati individuati diversi tipi di requisiti e si è quindi fatto utilizzo di un codice identificativo per distinguerli.
Il codice dei requisiti, dove ogni requisito è identificato con il carattere **R**, è così strutturato:

F: Funzionale.

Q: Qualitativo.

V: Di vincolo.

N: Obbligatorio (necessario).

D: Desiderabile.

Nelle tabelle 4.1, 4.2 e 4.3 sono riassunti i requisiti e il loro tracciamento con gli use case delineati in fase di analisi.

4.5 Tabelle dei requisiti

Requisito	Descrizione	Use Case
RFN-1	La funzionalità di autenticazione deve essere accessibile solo a Utenti Non Autenticati	UC0
Continua nella prossima pagina...		

Tabella 4.1 – Continuo della tabella

Requisito	Descrizione	Use Case
RFN-2	L'utente deve essere in grado di accedere alla pagina registrazione detta *signup*. Questa pagina deve dare la possibilità di inserire i campi *Name, Surname, Email, Password, Confirm Password, Company Name, Business Type, Address, Phone Number, Contry, Language* e scelte *checkbox* GDPR e consenso Marketing	UC0.1
RFN-2.1	Nel *form* per la registrazione devono esserci filtri per il controllo dei dati immessi come *input*	UC0.1
RFN-2.2	Una volta immessi i dati nel *form* della pagina di registrazione questa deve mostrare un messaggio che indica al utente di attivare l'account con la conferma della *email*	UC0.1
RFD-2.2.1	Una volta confermata la email il processo di registrazione è completato, è richiesto che la app faccia automaticamente il *login* con l'account appena creato, quindi ad ogni registrazione avviene il *login* automatico	UC0.1
RFN-3	Nella pagina di *login* l'utente presenta le sue credenziali nel apposito *form*, cioè *email e password* per poter diventare un Utente Autenticato	UC0.2
RFN-3.1	Nella pagina di *login* nel caso in cui l'utente abbia inserito le credenziali non valide deve apparire un messaggio di errore	UC0.2

Continua nella prossima pagina...

Tabella 4.1 – Continuo della tabella

Requisito	Descrizione	Use Case
RFN-4	Deve esistere una pagina collegata alla pagina di *login* che permetta di fare il processo di ripristino *password*, detto *Recover Password*, con questo processo l'utente inserisce la propria *email* e dopo un processo di verifica gli viene confermato il ripristino della *password*	UC0.3
RFN-5	La funzionalità di Prodotti definisce la *feature* di visualizzazione di prodotti di Unox, sono tutti i prodotti esistenti e sono oggetti astratti da catalogo, non macchine reali, è richiesto solo la visualizzazione di un singolo prodotto nella sua *Product Page*	UC1
RFN-5.1	Un prodotto deve essere identificato dal proprio Product Code$_G$, se lo si identifica solo con il *product code* si intende che si vuole visualizzare il prodotto con l'ultimo ECN$_G$	UC1
RFN-5.2	Nel caso in cui si identifichi un prodotto tramite il suo seriale, si intende che si vuole una istanza di un prodotto, quindi un prodotto fisico realmente esistente. Questa metodologia permette di identificare un prodotto con un *ecng* specifico.	UC1
Continua nella prossima pagina...		

Tabella 4.1 – Continuo della tabella

Requisito	Descrizione	Use Case
RFN-5.3	Nella pagina visualizzazione prodotto deve essere possibile visualizzare informazioni riguardanti le specifiche del prodotto, i documenti al prodotto collegati ed i *files* scaricabili come Firmware$_G$ o modelli 3D.	UC1
RFN-6	Deve essere implementata la *feature* Service, in particolare Serviced Oven$_G$, è richiesto che l'utente visualizzi un singolo suo prodotto identificato con un seriale	UC2
RFN-6.1	Nella pagina di *Serviced Oven* si devono poter visualizzare le informazioni riguardanti la connessione, si deve poter avere riferimenti alla pagina Prodotto del forno, e poter visualizzare e modificare le informazioni aggiuntive legate al prodotto, in particolare informazioni legate al cliente presso cui il forno è installato.	UC2
RFN-7	Per tutte le piattaforme deve essere realizzato un sistema di navigazione che permetta di cambiare le sezioni della app	UC0, UC1, UC2
RFN-7.1	Per le piattaforme mobile è prevista la realizzazione di una Tab Bar$_G$ invece per la piattaforma *web* è prevista la realizzazione di una Nav Bar$_G$	UC0, UC1, UC2

Tabella 4.1: Tabella del tracciamento dei requisiti funzionali.

Requisito	Descrizione	Use Case
RQN-1	La navigazione della app deve essere funzionale sia da *mobile* che da *web*, permettendo di indentificare con facilità il corretto flusso di funzionamento della *app* indipendentemente dalla piattaforma in uso.	-
RQN-2	Il codice prodotto deve essere scalabile e mantenibile, rispettando i *design pattern* attualmente già utilizzati nel ecosistema Unox.	-
RQN-3	Tutte le funzionalità realizzate devono essere testate su tutte le piattaforme di destinazione garantendo il loro funzionamento.	-

Tabella 4.2: Tabella del tracciamento dei requisiti qualitativi.

Requisito	Descrizione	Use Case
RVN-1	Le pagine e i componenti della applicazione devono essere realizzati tenendo conto delle indicazioni date dal team di *Design*, rispettando un Design System$_G$ esistente e le convenzioni date per stili grafici compresi *UI/UX*	-
RVN-2	Le *feature* realizzate per la applicazione devono essere compatibili con tutte le piattaforme di destinazione.	-

Tabella 4.3: Tabella del tracciamento dei requisiti di vincolo.

Capitolo 5

Progettazione e Codifica

In questo capitolo verranno descritti i processi di progettazione e codifica utilizzati nello sviluppo dell'applicazione DDC Service$_G$. Si esamineranno l'architettura dell'applicazione, le tecnologie utilizzate per il *frontend* e il *backend*, i protocolli di comunicazione, l'autenticazione e l'architettura a componenti.

5.1 *Backend* e *frontend*

In questa sezione, verranno descritti il *backend* e il *frontend* dell'applicazione DDC Service$_G$. Si analizzeranno le differenze tra le due parti, le tecnologie utilizzate e le responsabilità di ciascuna componente.

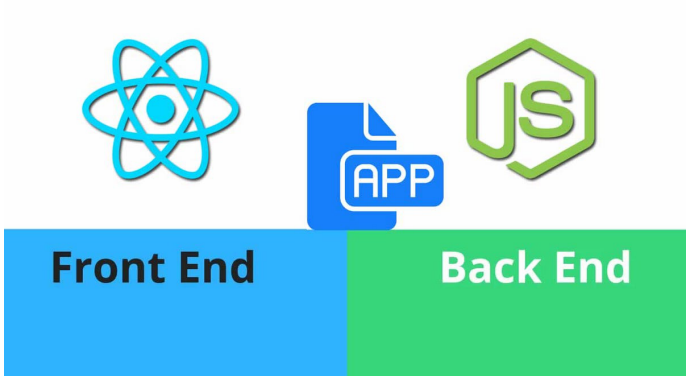


Figura 5.1: App moderne: Frontend e Backend

5.1.1 Responsabilità del *backend*

Il *backend* dell'applicazione è scritto in *Node.js* utilizzando *TypeScript*, una scelta che combina la flessibilità e la velocità di *Node.js* con la robustezza della tipizzazione statica di *TypeScript*. Il *backend* ha diverse responsabilità cruciali:

- **Gestione dei Dati**: Il *backend* gestisce l'archiviazione, il recupero e la manipolazione dei dati. Utilizza diversi *Database* per memorizzare informazioni strutturate. Inoltre, il *backend* si interfaccia con servizi di *cloud storage* come *Amazon S3* di *AWS* per memorizzare file e oggetti di grandi dimensioni, garantendo alta disponibilità e durabilità.
- **Autenticazione**: L'autenticazione degli utenti è un'altra responsabilità fondamentale del *backend*. L'autenticazione gestita dal *backend* offre un livello di sicurezza maggiore per diverse ragioni:
  - **Controllo Centralizzato:** Il *backend* funge da punto centrale per la verifica delle credenziali degli utenti, garantendo che tutti i tentativi di accesso siano gestiti in modo uniforme e sicuro. Questo riduce il rischio di inconsistenze e vulnerabilità che potrebbero sorgere se l'autenticazione fosse distribuita su più *client*.
  - **Protezione dei Segreti:** Nel *backend*, è possibile mantenere i segreti e le chiavi di crittografia al sicuro, lontano dai dispositivi degli utenti dove potrebbero essere più facilmente compromessi. Ciò include la gestione sicura delle *password*, che vengono memorizzate come *hash* sicuri, e la generazione di *token* di accesso.
  - **Validazione e Scadenza dei Token:** Il *backend* può gestire la generazione, la validazione e la scadenza dei *token* di accesso *JWT*, assicurando che solo i token validi e non scaduti possano essere utilizzati per accedere alle risorse protette. Questo meccanismo previene l'accesso non autorizzato e garantisce che gli utenti debbano autenticarsi periodicamente.
  - **Log e Monitoraggio:** Il *backend* può registrare tutte le attività di autenticazione, facilitando il monitoraggio e l'analisi dei tentativi di accesso.

Questo aiuta a identificare e rispondere rapidamente a potenziali attacchi, come tentativi di forza bruta o accessi sospetti.

Questi vantaggi fanno sì che l'autenticazione gestita dal *backend* sia un componente cruciale per la sicurezza complessiva dell'applicazione, proteggendo le informazioni sensibili degli utenti e garantendo l'integrità e la riservatezza dei dati.

- **Logica di Business**: Il *backend* contiene la logica di business dell'applicazione, questa logica è separata in moduli e servizi, rendendo il codice più organizzato e manutenibile. I servizi nel *backend* sono responsabili di eseguire operazioni come la validazione dei dati, l'elaborazione delle transazioni, e la comunicazione con servizi esterni.
- **API**: Il *backend* espone *API* REST e *GraphQL* che permettono al *frontend* di interagire con i dati e le funzionalità dell'applicazione. Le *API REST* sono utilizzate per operazioni standard *CRUD (Create, Read, Update, Delete)* e seguono una struttura basata su risorse. *GraphQL*, d'altra parte, offre una maggiore flessibilità permettendo al *frontend* di specificare esattamente quali dati necessita, riducendo così il sovraccarico di rete e migliorando l'efficienza.

5.1.2 Responsabilità del *frontend*

Il *frontend* dell'applicazione è la parte visibile e interattiva che gli utenti utilizzano. È sviluppato utilizzando *Expo* e *Next.js*, con *Expo* destinato alle piattaforme *mobile* e *Next.js* alle piattaforme *web*. Il linguaggio di programmazione utilizzato per il *frontend* è *Typescript*. Le principali responsabilità del *frontend* includono:

- **Interfaccia Utente (UI)**: Il *frontend* è responsabile della presentazione visiva e dell'interazione dell'utente con l'applicazione. Utilizza componenti *React* per costruire un'interfaccia modulare e riutilizzabile. Ogni componente rappresenta una parte della *UI*, come bottoni, moduli di *input*, e *layout* di pagina.
- **Gestione dello Stato**: La gestione dello stato è un aspetto critico del *frontend*. Utilizzando *Redux*, una libreria per la gestione dello stato, l'applicazione mantiene uno stato globale che può essere condiviso tra vari componenti.

- **Interazione con le *API***: Il *frontend* interagisce con il *backend* attraverso le *API REST* e *GraphQL*. Utilizzando librerie come *Axios* o *Fetch*, il *frontend* invia richieste *HTTP* al *backend* per recuperare, creare, aggiornare o eliminare dati. Le risposte del *backend* sono quindi utilizzate per aggiornare l'interfaccia utente, rendendo i dati dinamicamente disponibili agli utenti.

- **Navigazione e Routing**: Il *frontend* gestisce la navigazione tra le diverse pagine dell'applicazione, permettendo agli utenti di spostarsi fluidamente all'interno dell'interfaccia. Questo include la definizione di percorsi e la gestione delle transizioni tra le pagine, assicurando una suddivisione logica dell'applicazione in sezioni distinte e accessibili.

5.1.3 Differenze tra *backend* e *frontend*

Sebbene il *backend* e il *frontend* siano strettamente collegati, essi hanno ruoli e responsabilità distinti:

- **Tecnologie**: Il *backend* è sviluppato in *Node.js*, focalizzandosi sulla gestione dei dati, la logica di business e le *API*. Il *frontend*, invece, è sviluppato utilizzando *Expo* per le piattaforme *mobile* e *Next.js* per le piattaforme *web*, concentrandosi sull'interfaccia utente e l'esperienza dell'utente.

- **Responsabilità**:

  - **Backend**: Gestisce la logica di business, l'autenticazione, la gestione dei dati e la comunicazione con i servizi esterni.

  - **Frontend**: Si occupa della presentazione visiva, l'interazione dell'utente, la gestione dello stato e la navigazione.

- **Interazione**: Il *backend* e il *frontend* comunicano tramite *API*, dove il *backend* fornisce i dati e le funzionalità necessarie, mentre il *frontend* li utilizza per creare un'interfaccia utente interattiva e dinamica.

5.1.4 Flusso di Lavoro Complessivo

Il flusso di lavoro dell'applicazione prevede che l'utente interagisca con il *frontend*, che a sua volta invia richieste al *backend*. Il *backend* elabora queste richieste, esegue la logica di business necessaria, interagisce con i *database* e i servizi esterni, e infine restituisce una risposta al *frontend*. Il *frontend* quindi aggiorna l'interfaccia utente in base alla risposta ricevuta, garantendo un'esperienza utente fluida e interattiva. In sintesi, il *backend* e il *frontend* dell'applicazione DDC Service$_G$ lavorano insieme per fornire un servizio completo ed efficiente, ognuno con ruoli e responsabilità specifici ma complementari. Questa architettura modulare e separata permette di sviluppare, mantenere e scalare l'applicazione in modo efficace.

5.1.5 Struttura delle Applicazioni

La struttura complessiva delle applicazioni è organizzata in modo da mantenere il codice manutenibile e modulare. Questo approccio consente di gestire facilmente la complessità del progetto, facilitando lo sviluppo, la collaborazione e la scalabilità. La disposizione delle *directory* e dei file è progettata per riflettere la suddivisione logica delle funzionalità, garantendo che ogni componente del sistema sia isolato e riutilizzabile.

5.1.6 Monorepo

L'utilizzo di una monorepo$_G$ è una strategia che permette di gestire tutto il codice del progetto in un unico *repository*. Questo approccio offre numerosi vantaggi, tra cui:

- **Condivisione del Codice**: Facilita la condivisione di moduli e librerie comuni tra diverse parti dell'applicazione, evitando duplicazioni e incoerenze. Questo approccio è particolarmente utile per il progetto *DDC Service*, poiché consente di riutilizzare parti comuni tra l'applicazione *DDC* e *DDC Service*, come il *Design System*, garantendo una coerenza visiva e funzionale tra le due applicazioni.

- **Gestione delle Dipendenze**: Permette di avere una gestione migliore delle dipendenze, semplificando l'aggiornamento e la sincronizzazione delle librerie utilizzate. Questo è essenziale per mantenere un ambiente di sviluppo stabile e aggiornato, riducendo i conflitti e le incompatibilità tra le diverse parti del progetto.

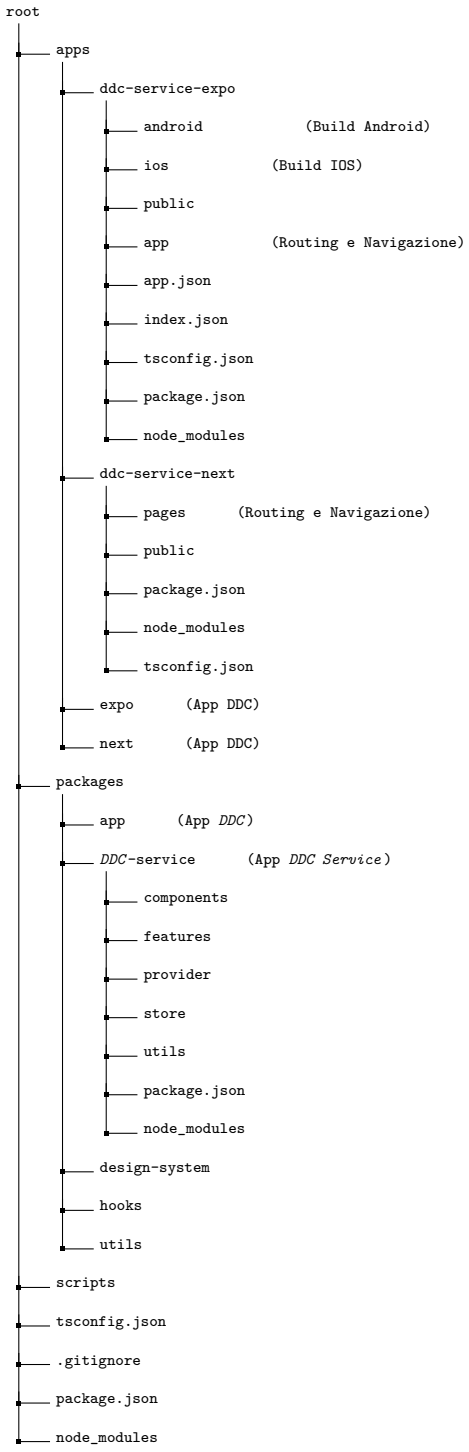
- **Collaborazione**: Migliora la collaborazione tra i *team*, offrendo una visione unificata del progetto e facilitando la revisione del codice. I team possono lavorare contemporaneamente su diverse parti dell'applicazione, beneficiando della trasparenza e della coerenza del codice condiviso.

- **Strumenti di Build e CI/CD**: Consente di configurare strumenti di build e *pipeline CI/CD* in modo centralizzato, ottimizzando i processi di integrazione e distribuzione continua. Questo permette di automatizzare i test, il rilascio e il *deploy* delle applicazioni *DDC* e *DDC Service*, assicurando che le nuove funzionalità e le correzioni di bug siano implementate rapidamente e senza intoppi.

Per questo progetto si è deciso di utilizzare la tecnologia *monorepo* presente con *npm workspaces*. Questo primo approccio è stato scelto in quanto è stata la configurazione già addottata per *DDC*, tuttavia, come verrà analizzato nel capitolo successivo, ci sono soluzioni migliore.

5.1.6.1 Struttura delle *directory*

La struttura del progetto è organizzata come segue:



5.1.7 Visione Generale

La struttura del progetto è organizzata per supportare lo sviluppo e la gestione di quattro applicazioni principali: due applicazioni *DDC* (una per *Expo* e una per *Next*)

e due applicazioni *DDC Service* (una per *Expo* e una per *Next*). La suddivisione in diverse *directory* facilita la condivisione del codice, delle risorse e delle configurazioni tra le applicazioni, migliorando la manutenibilità e la modularità del progetto.[1]

Le *directory* principali sono suddivise come segue:

- **apps**: Contiene le applicazioni specifiche per *DDC* e *DDC Service*. Ogni applicazione ha una *directory* separata per le versioni *Expo* e *Next*. Le applicazioni contenute qui sono la struttura portante rispettivamente di *Next* ed *Expo*, e il codice contenuto dentro queste cartelle si occupa solamente di navigazione e *routing*.
- **packages**: Contiene le applicazioni ma anche i pacchetti condivisi tra le applicazioni, inclusi componenti, *utils, hook* e il *Design System*. il package utils contiene funzioni utili per tutte le applicazioni che possono essere riutilizzate in ambito sviluppo multipiattaforma.
- **scripts**: Contiene script per automatizzare varie attività di sviluppo e di *deployment*.
- **Configurazioni**: File di configurazione globali, come *tsconfig.json*, *package.json* e `.gitignore`.

5.1.8 Descrizione delle *directory*

- **apps** La *directory* `apps` contiene le applicazioni specifiche per *DDC* e *DDC Service*, organizzate come segue:
  - **ddc-service-expo**: Contiene la versione *Expo* dell'applicazione *DDC Service*. Include le *directory* per Android e iOS, oltre a file di configurazione specifici per *Expo*.
  - **ddc-service-next**: Contiene la versione *Next* dell'applicazione *DDC Service*. Include le pagine dell'applicazione e i file di configurazione specifici per *Next.js*.

[1] *Migrating to a Monorepo Using NPM 7 Workspaces*. URL: https://medium.com/edgybees-blog/how-to-move-from-an-existing-repository-to-a-monorepo-using-npm-7-workspaces-27012a100269.

  - **expo (DDC)**: Contiene la versione *Expo* dell'applicazione *DDC*.
  - **next (DDC)**: Contiene la versione *Next* dell'applicazione *DDC*.

- **packages** La *directory* `packages` contiene i pacchetti condivisi tra le applicazioni, organizzati come segue:
  - **app**: Pacchetto contenente le applicazioni e le pagine condivise tra le versioni *Expo* e *Next* dell'applicazione *DDC*.
  - **ddc-service**: Pacchetto contenente componenti, funzionalità, provider, *store* e utilità specifici per l'applicazione *DDC Service*.
  - **design-system**: Pacchetto contenente il *Design System* condiviso tra le applicazioni, incluso il tema dell'applicazione e i componenti di base.
  - **hooks**: Pacchetto contenente *hook* condivisi tra le applicazioni.
  - **utils**: Pacchetto contenente utilità condivise tra le applicazioni.

- **scripts** La *directory* `scripts` contiene script per automatizzare varie attività di sviluppo, come l'installazione delle dipendenze, la costruzione e il deployment delle applicazioni.

- **Configurazioni** Nella root del progetto si trovano vari file di configurazione globali, tra cui:
  - `postinstall.json`: Script eseguiti dopo l'installazione delle dipendenze.
  - `tsconfig.json`: Configurazione *Typescript* condivisa tra tutte le applicazioni e i pacchetti.
  - `.gitignore`: File che specifica i file e le *directory* da ignorare nel controllo di versione.
  - `package.json`: Gestione delle dipendenze e script di progetto.
  - `node_modules`: *directory* generata automaticamente che contiene tutte le dipendenze installate.

5.1.9 Gestione delle Dipendenze

All'interno del file *package.json* situato nella *root* del progetto, sono configurati diversi parametri che definiscono la struttura della *monorepo*. In particolare, c'è una suddivisione tra due principali categorie di *workspaces*: *apps* e *packages*. Per *workspace* ci si riferisce alle cartelle contenute all'interno delle *directory apps* e *packages*. Ad esempio, *ddc-service-expo* è considerato un *workspace*.
Questa suddivisione permette una gestione organizzata e modulare del codice, facilitando la manutenzione e la scalabilità del progetto. Per quanto riguarda la gestione delle dipendenze, quando si utilizza un *package* esterno, è necessario installarlo nel corretto *workspace*. Ogni *workspace* ha il proprio file *package.json*, che contiene solo le dipendenze specifiche necessarie per quel *workspace*. Questa separazione garantisce che ogni parte del progetto abbia accesso solo alle librerie di cui ha bisogno, evitando conflitti e riducendo il rischio di dipendenze non utilizzate. Inoltre, ogni *workspace* ha una propria cartella *node_modules* dedicata, che contiene tutte le librerie e le dipendenze installate specificamente per quel *workspace*. Questa configurazione consente di mantenere un ambiente di sviluppo isolato per ciascun *workspace*, facilitando la gestione e l'aggiornamento delle dipendenze in modo indipendente. In sintesi, l'uso di *workspaces* e la gestione decentralizzata delle dipendenze attraverso file *package.json* separati e cartelle *node_modules* dedicate contribuiscono a rendere la *monorepo* più organizzata, modulare e manutenibile. Questa struttura permette ai *team* di sviluppo di lavorare in modo più efficiente, riducendo i tempi di integrazione e semplificando il processo di aggiornamento delle librerie.[2]

[2] *Migrating to a Monorepo Using NPM 7 Workspaces*.

5.2 Comunicazione

La comunicazione tra il *frontend* e il *backend* è un aspetto cruciale nello sviluppo di applicazioni moderne. Questo capitolo descrive i protocolli di comunicazione utilizzati, come *REST e GraphQL*, e l'uso di strumenti di *code generation* per mantenere il codice tipizzato e sincronizzato con lo schema *GraphQL*.

5.2.1 Protocolli di Comunicazione

Per garantire una comunicazione efficace e sicura tra il *frontend* e il *backend*, sono stati adottati diversi protocolli di comunicazione. Tra i principali protocolli utilizzati troviamo *REST e GraphQL*, ciascuno con specifici casi d'uso e vantaggi.

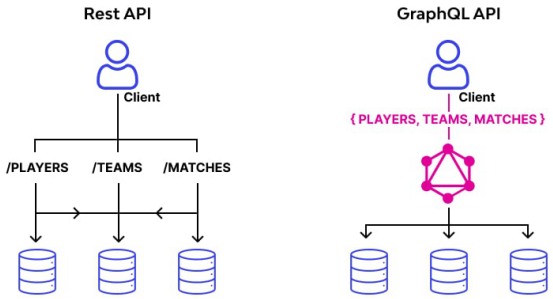


Figura 5.2: Rest API vs GraphQL API

5.2.1.1 REST

La *API REST* è stata utilizzata principalmente per l'autenticazione degli utenti. *REST* è un'architettura leggera che utilizza i metodi *HTTP standard (GET, POST, PUT, DELETE)* per eseguire operazioni *CRUD (Create, Read, Update, Delete)* sui dati. Un esempio di *endpoint REST* per l'autenticazione potrebbe essere:

```
POST /api/auth/login
{
    "username": "example_user",
    "password": "example_password"
}
```

Questo *endpoint* accetta le credenziali dell'utente e, se valide, restituisce un *token JWT* che viene utilizzato per autenticare le richieste successive.

5.2.1.2 GraphQL

GraphQL è un linguaggio di *query* per le *API* che offre una maggiore flessibilità rispetto a *REST*. In *DDC Service*, una volta ottenuto il *token JWT* tramite *REST*, *GraphQL* viene utilizzato per interrogare il *backend*. I vantaggi di *GraphQL* includono la possibilità di richiedere esattamente i dati necessari e la riduzione del numero di richieste necessarie per ottenere tutte le informazioni richieste.

Esempio generico di una *query GraphQL*:

```
query {
    user(id: "123") {
        id
        name
        email
    }
}
```

Esempio generico di una mutazione *GraphQL*:

```
mutation {
    updateUser(id: "123", input: {name: "New Name"}) {
        id
        name
        email
    }
}
```

GraphQL è stato scelto per la sua efficienza nella gestione delle richieste complesse e per la sua capacità di evolversi senza influenzare le versioni precedenti dell'*API*.

5.2.2 GraphQL Codegen e RTK *query*

L'utilizzo combinato di strumenti di *code generation* per *GraphQL*, come *GraphQL Code Generator*, e Redux Toolkit$_{G}$ è fondamentale per mantenere il codice tipizzato e sincronizzato con lo schema *GraphQL*. *GraphQL Code Generator* genera automaticamente tipi *Typescript* per le *query*, le mutazioni e i frammenti definiti, mentre *RTK query* facilita l'integrazione di queste *query* nel sistema di gestione dello stato. Questa combinazione migliora la sicurezza del tipo e riduce gli errori di *runtime*, oltre a fornire un'architettura chiara e scalabile per la gestione delle *API GraphQL* nel *frontend*.

5.2.2.1 Configurazione e Utilizzo

La configurazione di *GraphQL Codegen* nel progetto è semplice e diretta. I file *.graphql*, contenenti tutte le *query* da eseguire, sono memorizzati all'interno della cartella *graphql/documents*.

Di seguito è riportato un esempio di configurazione nel file *codegen.config.ts*:

```
import { CodegenConfig } from '@graphql-codegen/cli'

const config: CodegenConfig = {
schema: './store/graphql/schema.json',
documents: ['./store/graphql/documents/**/*.graphql'],
ignoreNoDocuments: true, // for better experience with the watcher
generates: {
    './store/graphql/queryBaseApi.ts': {
    plugins: [
        'typescript',
        'typescript-operations',
        'typescript-resolvers',
        {
        'typescript-rtk-query': {
            importBaseApiFrom: 'ddc-service/store/api/baseApi',
            importBaseApiAlternateName: 'baseApi',
            exportHooks: true,
        },
        },
    ],
    },
},
}

export default config
```

Codice 5.1: Configurazione *GraphQL Codegen: codegen.config.ts*

Questo file di configurazione specifica l'endpoint dello schema *GraphQL* (*schema.json*), i documenti *GraphQL* (*documents*) da cui generare il codice, e i plugin da utilizzare per la generazione. Inoltre, viene generato un file chiamato *queryBaseApi.ts*, che contiene tutte le *query* e le mutazioni definite nei file *.graphql*.

Una volta configurato, è possibile eseguire il comando di generazione:

`graphql-codegen`

Il file *queryBaseApi.ts* generato viene utilizzato per integrare le *query* e le mutazioni *GraphQL* nel codice del *frontend*, sfruttando *Redux* per la gestione della *cache*.

Queste *API* sono cachate su *Redux* e possono essere chiamate semplicemente con:

```
import { useGetOvensQuery }
from 'ddc-service/store/graphql/queryBaseApi'
```

Per utilizzare la *query*:

```
const { data, isFetching, isLoading, isSuccess,
isError, isUninitialized, error, refetch } = useGetOvensQuery({})
```

Codice 5.2: Utilizzo *API GraphQL*

- **data**: contiene i dati recuperati dalla *query*.
- **isFetching**: booleano che indica se la *query* è attualmente in fase di recupero.
- **isLoading**: booleano che indica se la *query* è in fase di caricamento iniziale.
- **isSuccess**: booleano che indica se la *query* è stata completata con successo.
- **isError**: booleano che indica se c'è stato un errore nell'esecuzione della *query*.
- **isUninitialized**: booleano che indica se la *query* non è stata ancora eseguita.
- **error**: contiene informazioni sull'errore se la *query* ha fallito.
- **refetch**: funzione che permette di eseguire nuovamente la *query*.

L'utilizzo di *GraphQL Codegen* garantisce che il codice rimanga sincronizzato con lo schema *GraphQL*, riducendo il rischio di errori e migliorando la produttività del team di sviluppo.

5.2.3 Autenticazione

Il sistema di autenticazione utilizzato nell'applicazione è cruciale per garantire che solo gli utenti autorizzati possano accedere ai dati e alle funzionalità disponibili. Questo sistema si basa su flussi di autenticazione ben definiti, utilizza JSON Web Token$_G$ e implementa diverse strategie di sicurezza per mantenere l'integrità e la riservatezza dei dati.

5.2.3.1 Backend Stateless

Un *backend stateless* è un'architettura in cui il *server* non mantiene alcuna informazione di stato tra le richieste dei *client*. Ogni richiesta che arriva al *server* deve contenere tutte le informazioni necessarie per essere processata. Questo approccio offre diversi vantaggi, tra cui:

- **Scalabilità**: Poiché il *server* non deve mantenere informazioni di stato, può facilmente scalare orizzontalmente aggiungendo più istanze per gestire un numero maggiore di richieste.
- **Resilienza**: Se un *server* va *offline*, le richieste possono essere indirizzate a un altro *server* senza perdere dati di sessione.
- **Semplicità**: La gestione delle sessioni non è necessaria, riducendo la complessità del codice del *server*.

Nel contesto di questo progetto, il *backend* è ospitato su *AWS*, utilizzando vari servizi cloud per garantire affidabilità e scalabilità.

5.2.3.2 Flussi di Autenticazione

I flussi di autenticazione principali includono *login*, registrazione e recupero della *password*. Durante il *login*, il *frontend* invia le credenziali dell'utente al *backend* tramite un *endpoint REST*. Il *backend* verifica le credenziali e, se valide, genera un *token JWT* che viene inviato al *frontend* per essere utilizzato nelle richieste successive. Nel caso della registrazione, l'utente fornisce le informazioni necessarie, che vengono inviate al *backend*. Il *backend* crea un nuovo utente nel database e, se

la registrazione è avvenuta con successo, restituisce un token JWT al *frontend*. Per il recupero della password, l'utente invia una richiesta di *reset* tramite un *endpoint REST*. Il *backend* genera un link di recupero della password e lo invia all'indirizzo email dell'utente. L'utente può quindi utilizzare questo *link* per reimpostare la propria *password*.

5.2.3.3 Gestione Cookie

Inizialmente, il *backend* utilizzava *Passport* per gestire le sessioni in modalità *stateless*, il che significa che non memorizza informazioni sulla sessione degli utenti tra le richieste. In questo approccio, i cookie venivano inviati al client con il comando *Set-Cookie*, e il *client* era responsabile di memorizzare e inviare i *cookie* contenenti le sessioni e le firme con ogni richiesta successiva. Tuttavia, la gestione dei *cookie* su piattaforme native come *Android* e *IOS* ha presentato delle sfide. Ad un primo sguardo può sembrare impossibile avere i *cookie* su piattaforme native, e gli si considera solo per il *web*. In verità *React Native* e la sua libreria *fetch* integrata hanno un'implementazione nativa dei *cookie*. Questa implementazione nativa innanzitutto diferisce in base alla piattaforma, tra *IOS e Android*, e ha diverse politiche di gestione rispetto al browser, per esempio la gestione differisce nelle politiche *CORS*. Inoltre, la gestione dei *cookie* su dispositivi mobili richiede l'uso di librerie di gestione dei *cookie* e non può essere effettuata tramite semplici richieste *HTTP* come avviene nei browser. Queste limitazioni hanno reso chiaro che, pur essendo possibile, un sistema di autenticazione basato sui cookie per piattaforme native non è praticabile.[3]

5.2.3.4 Gestione dei Token JWT

A causa delle problematiche riscontrate con i cookie, è stato deciso di adottare i JSON *web* Tokens (JWT) per l'autenticazione. I JWT sono token compatti, URL-safe, che rappresentano in modo sicuro le informazioni tra due parti. Un token JWT è composto da tre parti: header, payload e signature.

[3] *React Native Cookie Authentication.* URL: https://javascript.plainenglish.io/react-native-cookie-authentication-83ef6e84ba70.

- **Header**: Contiene il tipo di token (JWT) e l'algoritmo di firma (es. HMAC SHA256).
- **Payload**: Contiene le informazioni dell'utente (claims), come l'ID utente e le autorizzazioni.
- **Signature**: Viene creata utilizzando l'algoritmo specificato nell'header e una chiave segreta. Serve a verificare che il token non sia stato modificato.

I JWT sono sicuri perché la firma digitale impedisce la manipolazione dei dati contenuti nel token. Inoltre, essi funzionano bene con un *backend* stateless, poiché tutte le informazioni necessarie per autenticare una richiesta sono contenute nel token stesso. Questo elimina la necessità di mantenere sessioni sul *server*, migliorando la scalabilità e riducendo il carico sul *backend*. In conclusione, l'adozione dei JWT per l'autenticazione ha migliorato la sicurezza e l'efficienza dell'applicazione, risolvendo le problematiche di gestione dei cookie su piattaforme mobili e supportando un'architettura *backend* stateless.

5.3 Architettura a Componenti

5.3.1 Componenti di Base *Design System*

I componenti di base dell'applicazione sono gli elementi fondamentali utilizzati per costruire l'interfaccia utente. Questi componenti sono creati utilizzando *React* e fanno parte del *Design System*, un insieme di linee guida e componenti riutilizzabili che garantiscono coerenza e usabilità nell'applicazione.

Esempi di componenti di base includono *Button*, *input*, *Text*, etc... Di seguito è riportato un esempio di un componente *Text* a scopo dimostrativo:

```
import \textit{React} from 'react';
import { Text as RNText, TextProps } from 'dripsy';
import { useTheme } from '../Theme.tsx';

const Text: React.FC<TextProps> = ({ style, ...props }) => {
    const { colors } = useTheme();
    return <RNText style={[{ color: colors.text }, style]} {...props} />;
};

export default Text;
```

Codice 5.3: Esempio Componente Text *Design System*

Il codice sopra è solo un esempio di come potrebbe essere implementato un componente di base. Un componente di base è un componente astratto che include solo funzionalità dedicate a se stesso, senza logica legata all'architettura dell'applicazione. Ad esempio, un componente del *Design System* dedicato a elementi di *layout* o a elementi comuni dell'interfaccia utente dell'applicazione è un componente di base. Un errore comune nella realizzazione di componenti di base del *Design System* è introdurre logica specifica dell'applicazione, come la gestione degli stati tramite *Redux*. Questa pratica lega la logica dell'applicazione ai componenti di base, rendendoli non riutilizzabili in futuro. I componenti di base dovrebbero essere il più astratti possibile, gestendo la propria logica internamente e ricevendo i parametri di *input* come props. Se progettati correttamente, questi componenti possono essere utilizzati in diverse parti dell'applicazione senza dipendere da contesti specifici. Per esempio, se un componente di base del *Design System* viene utilizzato in una funzionalità dell'applicazione, ci si aspetta che possa essere riutilizzato anche in altre funzionalità senza modifiche. Questi componenti devono seguire le linee guida del *Design System* e le librerie grafiche comuni, ma devono rimanere indipendenti dalla logica specifica dell'applicazione. Seguendo queste linee guida, i componenti di base possono essere progettati in modo da essere riutilizzabili, mantenibili e indipendenti dalla logica dell'applicazione, contribuendo così a un codice più pulito e modulare.

5.3.2 Componenti Compositi

I componenti compositi combinano i componenti di base per formare parti più complesse dell'interfaccia utente. Ad esempio, un componente *Form* potrebbe combinare vari *input* e bottoni per creare un modulo di inserimento dati.

Questi componenti compositi sono spesso legati a singole funzionalità dell'applicazione e sono costruiti utilizzando i componenti di base del *Design System*. La gestione gerarchica dei componenti in *React* segue il pattern MVVM (Model-View-ViewModel), dove:

- **Model**: rappresenta i dati dell'applicazione.
- **View**: rappresenta l'interfaccia utente e mostra i dati del Model.
- **ViewModel**: gestisce la logica di presentazione e l'interazione tra la View e il Model.

In *React*, i componenti della View sono implementati come componenti funzionali o classi, mentre il *ViewModel* può essere rappresentato da hook personalizzati o dallo stato locale dei componenti. Questo approccio è particolarmente utile in ambito native perché consente una gestione modulare e riutilizzabile dell'interfaccia utente.

Ecco un esempio di componente composito TitleSection che combina componenti di base del *Design System*:

```
import Button, { ButtonProps }
    from '@unox/design-system/src/components/Button'
import { View } from 'dripsy'
import { isWebPlatform, useIsMobile } from '@unox/utils/breakpoints'
import Text from '@unox/design-system/src/components/Text'

const TitleSection = (props: {
    title?: string
    size: 'small' | 'large'
    textAlignCenter?: boolean
    subtitle?: string
    backgroundColor?: string
    button?: { showOnDesktop?: boolean; buttonProps: ButtonProps }
    secondButton?: { showOnDesktop?: boolean; buttonProps: ButtonProps }
    }) => {
    const { size, title, textAlignCenter,
        subtitle, button, secondButton, backgroundColor } = props
    const isMobile = useIsMobile()
    const isSmall = size === 'small'
    const isLarge = size === 'large'

    return (/**/)
    //
}
```

Codice 5.4: Esempio Componente TitleSection *DDC Service*

In questo esempio, si può vedere come i componenti *Button, View e Text* vengano combinati per creare un nuovo componente composito. I componenti compositi uniscono componenti di base per formare parti più complesse dell'interfaccia utente. Questi componenti sono progettati per supportare funzionalità specifiche dell'applicazione, integrando spesso la logica necessaria per realizzare tali funzionalità. I componenti compositi prendono componenti di base dal *Design System* e li com-

binano per creare strutture più articolate come moduli, card o sezioni di *layout*. Questo approccio consente di costruire interfacce utente modulari e riutilizzabili, mantenendo il codice organizzato e facile da gestire.

5.3.3 Esempi di Componenti Compositi

In questa sezione verranno mostrati due esempi di componenti compositi. Verranno mostrati solo gli import di questi componenti, in quanto già da qui si possono capire le funzionalità dei componenti e la loro struttura.

5.3.3.1 SignInForm

```
import { useSx, View } from 'dripsy'
import { Button } from '@unox/design-system/'
import { Text } from '@unox/design-system/'
import InputField from '@unox/design-system/src/components/InputField'
import { useRouter } from 'solito/router'
import useAuth from '../useAuth'
import { TextLink } from 'solito/link'
import useStylesByVariants from '@unox/hooks/useStylesByVariants'
import { RootState } from 'ddc-service/store'
import { useSelector } from 'react-redux'
import { useEffect, useState } from 'react'
const SignInForm = () => {
    const [email, setEmail] = useState('')
    const [password, setPassword] = useState('')
    const { authSignIn } = useAuth()
    const AUTH_SIGNIN_ERROR =
        useSelector((state: RootState) => state.auth.AUTH_SIGNIN_ERROR)
    const { push, replace, back, parseNextPath } = useRouter()
    const errorModal = useErrorModal()
    const sx = useSx()
    const getStylesByVariants = useStylesByVariants()
    const recoverStyle = getStylesByVariants([['text.linkSmall']])
    const callBackSignIn = () => {
        authSignIn({ email, password })
    }
    return (/**/)
}
```

Codice 5.5: Esempio Componente SignInForm *DDC* Sercive

Il componente `SignInForm` è un modulo di autenticazione che permette agli

utenti di accedere all'applicazione inserendo le proprie credenziali. Analizziamo gli import per capire come è strutturato e quali funzionalità implementa:

- **Layout e Stile**
  - `useSx, View` da `dripsy`: Dripsy è una libreria per la gestione degli stili in *React* Native, particolarmente utile per applicazioni cross-platform. `useSx` viene utilizzato per definire stili inline, mentre `View` è un contenitore per *layout*.
  - `useStylesByVariants` da `@unox/hooks/useStylesByVariants`: Questo hook permette di gestire varianti di stili, rendendo più facile applicare stili diversi in base a determinate condizioni.
- **Componenti di Base del *Design System***
  - `Button, Text` da `@unox/design-system/`: Sono componenti di base per i pulsanti e il testo, forniti dal *Design System* dell'applicazione.
  - `InputField` da `@unox/design-system/src/components/InputField`: Un campo di *input* personalizzato, anch'esso parte del *Design System*, utilizzato per raccogliere dati dagli utenti.
- **Navigazione**
  - `useRouter` da `solito/router`: Un hook per gestire la navigazione all'interno dell'applicazione. `push`, `replace`, `back` e `parseNextPath` sono metodi per la gestione delle rotte.
  - `TextLink` da `solito/link`: Un componente per creare link testuali che permettono di navigare all'interno dell'applicazione.
- **Autenticazione**
  - `useAuth` da `../useAuth`: Un hook personalizzato per gestire l'autenticazione, contiene funzioni per il login, logout e altre operazioni correlate.
- **Gestione dello Stato**

  - `RootState` da `ddc-service/store`: Importa la definizione dello stato globale dell'applicazione gestito con *Redux*.
  - `useSelector` da `react-redux`: Un hook per accedere al stato *Redux*. In questo caso, viene utilizzato per ottenere l'errore di autenticazione (`AUTH_SIGNIN_ERROR`).

- **Effetti e Stato Locale**
  - `useEffect, useState` da `react`: Hook per gestire gli effetti collaterali (`useEffect`) e lo stato locale (`useState`) del componente. `useState` viene utilizzato per gestire gli *input* dell'email e della password.

- **Gestione degli Errori**
  - `useErrorModal` Hook per gestire e visualizzare errori tramite un modal.

Scopo del Componente

Il componente *SignInForm* è progettato per gestire il processo di accesso degli utenti. Gli utenti inseriscono le loro credenziali (email e password) nei campi di *input*. Quando l'utente preme il pulsante di accesso, la funzione *authSignIn* viene chiamata, utilizzando i dati di stato locali per tentare l'autenticazione. Il componente gestisce anche gli errori di autenticazione tramite *Redux* e permette di visualizzare un messaggio di errore e un modal se l'autenticazione fallisce. Inoltre, utilizza il router per navigare tra le schermate in base ai risultati dell'autenticazione.

5.3.3.2 ProductScreen

```
import { useGetOvensQuery, useGetProductDetailsQuery, useSearchProductQuery,
} from 'ddc-service/store/graphql/queryBaseApi'
import { RefreshControl } from 'react-native'
import { View, H1, ScrollView } from 'dripsy'
import { createParam } from 'solito'
import { useRouter } from 'solito/router'
import Breadcrumbs, { BreadcrumpsCtrl } from '../../components/Breadcrumbs'
import { isWebPlatform, useIsDesktop, useIsLargeDesktop, useIsMobile, useIsSmallDes
} from '@unox/utils/breakpoints'
import { SearchBanner } from '../../components/SearchBanner'
import TechnicalSpecsBand from './components/TechnicalSpecsBand'
import DownlaodArea from './components/DownloadArea'
import ItemSummaryCard from './components/ItemSummaryCard'
import ContentCTA from './components/ContentCTA'
import { usePathname } from 'solito/navigation'
import { useEffect } from 'react'
import productDetails from 'app/store/slices/productDetails'

export const ProductScreen = () => {
  const isDesktop = useIsDesktop()
  const isMobile = useIsMobile()
  const isTablet = useIsTablet()
  const isSmallDesktop = useIsSmallDesktop()
  const isLargeDesktop = useIsLargeDesktop()
  const { push, replace, back, parseNextPath } = useRouter()
  //Get Params, code from path and other from \textit{query}
  const { useParam } = createParam<{ code: string; ecn: string; serial: string }>()
  const [codeParam] = useParam('code')
  const [ecnParam] = useParam('ecn')
  const [serial] = useParam('serial')
  return (/*...*/)
}
```


Codice 5.6: Esempio Componente ProductScreen *DDC* Sercive

Il componente *ProductScreen* è una schermata complessa progettata per visualizzare i dettagli di un prodotto, inclusi specifiche tecniche, area di download, riepilogo e altro. Analizziamo gli import per capire come è strutturato e quali funzionalità implementa:

- **Query e Stato dei Dati**
  - `useGetOvensQuery, useGetProductDetailsQuery, useSearchProductQuery` da `ddc-service/store/graphql/queryBaseApi`: Questi hook vengono utilizzati per eseguire *query GraphQL* e recuperare i dati necessari dall'API, come dettagli del prodotto, risultati di ricerca e informazioni sui forni.
- **Componenti di *layout* e *UI***
  - `View, H1, ScrollView` da *dripsy*: *View* è un contenitore di *layout*, *H1* è un componente di intestazione e *ScrollView* permette lo scorrimento verticale del contenuto.
- **Navigazione e Routing**
  - `createParam` da *solito*: Una utility per creare e gestire i parametri delle URL o parametri ricevuti navigando nelle schermate della app nativa.
  - `useRouter` da *solito/router*: Un hook per gestire la navigazione all'interno dell'applicazione. *push*, *replace*, *back* e *parseNextPath* sono metodi per la gestione delle rotte.
  - `usePathname` da *solito/navigation*: Un hook per ottenere il pathname corrente della navigazione.
- **Componenti Personalizzati**
  - `Breadcrumbs, BreadcrumpsCtrl` da *../../components/Breadcrumbs*: Componenti per visualizzare e controllare i breadcrumb, facilitando la navigazione all'interno dell'applicazione.
  - `SearchBanner` da *../../components/SearchBanner*: Un componente per visualizzare un banner di ricerca.

  - `TechnicalSpecsBand` da *./components/TechnicalSpecsBand*: Un componente per visualizzare le specifiche tecniche del prodotto.
  - `DownloadArea` da *./components/DownloadArea*: Un componente per visualizzare l'area di download del prodotto.
  - `ItemSummaryCard` da *./components/ItemSummaryCard*: Un componente per visualizzare un riepilogo del prodotto.
  - `ContentCTA` da *./components/ContentCTA*: Un componente per visualizzare una call-to-action legata al contenuto.

- **Gestione della Responsività**
  - *isWebPlatform, useIsDesktop, useIsLargeDesktop, useIsMobile, useIsSmallDesktop, useIsTablet* da *@unox/utils/breakpoints*: Hook e utility per gestire la responsività dell'interfaccia utente su diverse piattaforme e dimensioni dello schermo.

- **Gestione dello Stato e degli Effetti**
  - `useEffect` da *react*: Un hook per gestire gli effetti collaterali nel componente. Utilizzato per eseguire azioni in risposta ai cambiamenti dello stato o delle props.
  - `productDetails` da *app/store/slices/productDetails*: Un slice di *Redux* per gestire lo stato dei dettagli del prodotto.

Scopo del Componente

Il componente *ProductScreen* è progettato per visualizzare una schermata dettagliata di un prodotto. Utilizza varie *query GraphQL* per recuperare i dati necessari e li visualizza tramite componenti personalizzati come *TechnicalSpecsBand*, *DownloadArea*, e *ItemSummaryCard*. La navigazione è gestita tramite *useRouter* e *createParam*, permettendo di passare parametri nella URL e navigare tra le schermate. Il componente è altamente responsivo, adattandosi a diverse dimensioni dello schermo grazie agli hook di breakpoints.

5.4 Gestione dello Stato

La gestione dello stato dell'applicazione è fondamentale per mantenere coerenza e prevedibilità nel comportamento dell'applicazione stessa. In questo progetto, si è utilizzato *Redux* per gestire lo stato globale. Redux è una libreria JavaScript per la gestione dello stato delle applicazioni, che offre un pattern architetturale predicibile e centralizzato. Con *Redux*, tutto lo stato dell'applicazione è mantenuto in un singolo *store*, che funge da unica fonte di verità.

Redux utilizza tre principi fondamentali:

- **Unico *store***: Lo stato dell'intera applicazione è memorizzato in un oggetto all'interno di un unico *store*.
- **Stato di Sola Lettura**: L'unico modo per modificare lo stato è emettere un'azione, un oggetto che descrive cosa è accaduto.
- **Modifiche con Funzioni Pure**: Per specificare come lo stato si trasforma a seguito delle azioni, si utilizzano pure functions chiamate riduttori.

5.4.1 Azioni e Riduttori

Le azioni sono oggetti che descrivono un cambiamento desiderato nello stato dell'applicazione. Un'azione ha sempre una proprietà *type*, che è una stringa che descrive il tipo di evento che si è verificato. Un'azione può anche avere un *payload* che contiene ulteriori informazioni necessarie per il cambiamento di stato.

Esempio di un'azione:

```
// actions.js
export const addItem = (item) => ({
    type: 'ADD_ITEM',
    payload: item
});
```

Codice 5.7: Esempio di Azione *Redux*

I riduttori (reducers) sono pure functions che specificano come lo stato dell'applicazione cambia in risposta a un'azione. I riduttori prendono lo stato corrente e l'azione come argomenti, e restituiscono un nuovo stato.

Esempio di un riduttore:

```
// reducer.js
const initialState = {
    items: []
};

const rootReducer = (state = initialState, action) => {
    switch(action.type) {
        case 'ADD_ITEM':
            return {
                ...state,
                items: [...state.items, action.payload]
            };
        default:
            return state;
    }
};

export default rootReducer;
```

Codice 5.8: Esempio di Riduttore *Redux*

5.4.2 Middleware

I *middleware* sono strumenti che estendono le capacità di *Redux* fornendo un punto di estensione tra il dispatch di un'azione e il momento in cui questa raggiunge il riduttore. I *middleware* possono essere utilizzati per gestire operazioni asincrone, *logging, routing,* e altro.

Uno dei middleware più comuni è *redux-thunk*, che permette di scrivere azioni che ritornano una funzione invece di un oggetto. Questa funzione può essere utilizzata per ritardare l'invio dell'azione, o per inviare solo se una certa condizione è vera.

Esempio di utilizzo di *redux-thunk*:

```
// actions.js
export const fetchItems = () => {
    return async (dispatch) => {
        dispatch({ type: 'FETCH_ITEMS_REQUEST' });
        try {
            const response = await fetch('/api/items');
            const data = await response.json();
            dispatch({ type: 'FETCH_ITEMS_SUCCESS', payload: data });
        } catch (error) {
            dispatch({ type: 'FETCH_ITEMS_FAILURE', payload: error });
        }
    };
};
```

Codice 5.9: Esempio di Azione Asincrona con *Redux*-thunk

Per utilizzare i *middleware* in *Redux*, si passa una funzione di applicazione del *middleware* al momento della creazione dello *store*.

Esempio di configurazione dello *store* con *middleware*:

```
// store.js
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducer';

const store = createStore(
    rootReducer,
    applyMiddleware(thunk)
);

export default store;
```

Codice 5.10: Esempio di Configurazione dello *store* con Middleware *Redux*

5.4.3 Gestione dello Stato con *Redux* Toolkit

Redux Toolkit è un pacchetto ufficiale consigliato da *Redux* per semplificare il modo in cui si scrivono logiche *Redux*, riducendo il boilerplate e migliorando la leggibilità del codice. Include diversi strumenti utili, tra cui un set di utility per creare azioni e riduttori, un middleware preconfigurato (Redux Thunk), e strumenti per gestire lo *store Redux* in modo più efficiente.

5.4.3.1 Caratteristiche Principali di *Redux* Toolkit

Le principali caratteristiche di *Redux* Toolkit includono:

- **createSlice**: Una funzione che accetta un oggetto di riduttori e azioni, generando automaticamente azioni e creatori di azioni, insieme a un riduttore.
- **configureStore**: Una funzione che combina la configurazione di *Redux*, inclusi riduttori, middleware e altri parametri, fornendo uno *store Redux* completo.
- **createAsyncThunk**: Una funzione per gestire azioni asincrone in modo chiaro e conciso.

5.4.3.2 Esempio di Utilizzo di *Redux* Toolkit

Ecco un esempio pratico di come *Redux Toolkit* viene utilizzato per gestire lo stato in un'applicazione:

```
// slices/counterSlice.ts
import { createSlice, PayloadAction } from '@reduxjs/toolkit';
interface CounterState { value: number; }
const initialState: CounterState = { value: 0, };
const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
        increment: (state) => {state.value++; },
        decrement: (state) => { state.value--; },
        reset: (state) => { state.value = 0; },
        incrementByAmount: (state, action: PayloadAction<number>) => {
            state.value += action.payload;
        },
    },
});
export const { increment, decrement,
    reset, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;
```

Codice 5.11: Esempio di slice con *Redux Toolkit*

Nell'esempio sopra, *counterSlice* definisce uno *slice* di stato con un riduttore e azioni correlate per gestire il contatore. Le azioni come *increment* e *decrement* sono create automaticamente da *createSlice*, insieme al riduttore che modifica lo stato.

Configuriamo quindi lo *store* utilizzando *configureStore*:

```
// store.ts
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './slices/counterSlice';

const store = configureStore({
    reducer: {
        counter: counterReducer,
    },
});

export default store;
```

Codice 5.12: Configurazione dello *store* con *Redux* Toolkit

In questo snippet, *configureStore* combina il riduttore *counterReducer* in uno *store Redux* completo. Lo *store* può quindi essere utilizzato nell'applicazione per gestire lo stato in modo centralizzato e prevedibile.

Redux Toolkit semplifica significativamente lo sviluppo e la manutenzione delle applicazioni *Redux*, offrendo uno strumento potente ma intuitivo per gestire lo stato. Utilizzando *createSlice*, *configureStore* e altri strumenti inclusi, *Redux Toolkit* promuove una pratica migliore nella gestione dello stato dell'applicazione.

5.5 Stilizzazione dei Componenti

La stilizzazione dei componenti in un'applicazione *React Native* è cruciale per garantire un'interfaccia utente coerente e gradevole su diverse piattaforme, inclusi dispositivi *mobile* e *web*. Le tecniche di stilizzazione devono essere modulari e riutilizzabili, in modo da semplificare lo sviluppo e il mantenimento del codice.

5.5.1 Esempi di Stilizzazione

Nel progetto *DDC Service*, la stilizzazione dei componenti è gestita principalmente attraverso *Dripsy*, una libreria che offre un'astrazione potente per la gestione

dello stile su piattaforme diverse. Dripsy utilizza una sintassi simile a *CSS-in-JS*, consentendo di definire stili in un modo dichiarativo e comodo.

Ecco un esempio di come *Dripsy* viene utilizzato per stilizzare un componente:

```
import { View } from 'dripsy';
const Button = () => {
    return (
        <View
            sx={{
                backgroundColor: 'primary',
                padding: 10,
                borderRadius: 5,
                alignItems: 'center',
            }}
        >
            <Text sx={{ color: 'white', fontSize: 16 }}>Press Me</Text>
        </View>
    );
};

export default Button;
```

Codice 5.13: Esempio di componente Button con Dripsy

Nell'esempio sopra, il componente *'Button* utilizza *View* da *Dripsy* per definire lo stile del pulsante. Il codice *sx* consente di specificare proprietà come *backgroundColor*, *padding*, *borderRadius* in un formato simile a *CSS*, rendendo facile aggiungere e modificare stili in modo dichiarativo.

5.5.2 Benefici di Dripsy in Ambito Cross-Platform

Dripsy si rivela particolarmente utile in ambito cross-platform per diversi motivi:

- **Sintassi Dichiarativa**: La sintassi simile a *CSS* di *Dripsy* facilita la definizione degli stili, riducendo la complessità e migliorando la leggibilità del codice.
- **Adattabilità su Diverse Piattaforme**: *Dripsy* è progettato per adattarsi automaticamente alle specifiche delle piattaforme target, come *IOS, Android e web*, consentendo uno sviluppo più efficiente e uniforme.
- **Riutilizzo dei Componenti**: Gli stili definiti con *Dripsy* possono essere riutilizzati senza modifiche significative tra le diverse piattaforme, promuovendo la coerenza visiva dell'applicazione.

L'utilizzo di *Dripsy* nel progetto *DDC Service* contribuisce quindi a una gestione efficace degli stili dei componenti, consentendo una implementazione fluida e uniforme su tutte le piattaforme supportate dall'applicazione.

5.6 Descrizione routing e navigazione

La navigazione all'interno delle applicazioni *web* e native presenta differenze sostanziali dovute alle diverse piattaforme e paradigmi di interazione. Su *web*, la navigazione avviene principalmente attraverso *URL* e percorsi, mentre su applicazioni native è più comune l'uso di *stack* di navigazione e transizioni tra schermate.

In questo progetto, si è utilizzato la libreria *Solito* per unificare la gestione del *routing* tra le piattaforme *Expo* e *Next.js*, riducendo la duplicazione del codice e semplificando la manutenzione. *Solito* consente di condividere la logica di navigazione tra *web* e *mobile*, permettendo una struttura di navigazione coerente e una gestione uniforme dei percorsi.

Routing tra le pagine

Il *routing* si riferisce al processo di determinazione della destinazione corretta in risposta alle interazioni dell'utente, come *clic* sui *link* o selezione di opzioni di menu. È stato implementato il *routing* utilizzando le convenzioni di *Expo* e *Next.js*, con una struttura a *folder* che facilita l'organizzazione e la scalabilità del progetto.

Navigazione con Solito

La scelta di utilizzare *Solito* è stata motivata dalla necessità di avere una soluzione che permettesse di condividere il codice tra le piattaforme *web* e *mobile*. Con *Solito*, è possibile definire la logica di navigazione una sola volta e utilizzarla sia su *Expo* che su *Next.js*. Ciò si traduce in un'esperienza utente più coerente e in una semplificazione dello sviluppo e della manutenzione delle applicazioni.

Struttura delle applicazioni

La struttura delle applicazioni segue una convenzione a *folder*, dove ogni cartella rappresenta una pagina o un gruppo di pagine correlate. Questo approccio è comune sia in *Expo* che in *Next.js* e, grazie a *Solito*, si è potuto sovrapporre queste strutture per mantenere la coerenza del codice e facilitare la navigazione *cross-platform* La struttura a *folder* consente di organizzare il codice in modo modulare e riutilizzabile, migliorando la leggibilità e la gestione del progetto.

Utilizzando queste tecniche, si può creare un sistema di navigazione che è efficace e intuitivo sia su piattaforme *web* che su dispositivi *mobile*, garantendo un'esperienza utente ottimale e una gestione efficiente del codice.

Capitolo 6

Studio fattibilità app in monorepo

6.1 Cos'è una monorepo

Una monorepo$_G$ è un singolo Repository che contiene più progetti, spesso correlati. Questo approccio consente di mantenere tutto il codice sorgente in un'unica posizione, facilitando la condivisione di codice e la gestione delle dipendenze. Una *monorepo* può contenere applicazioni Frontend$_G$ e Backend$_G$, librerie condivise e altre risorse comuni, permettendo una gestione centralizzata del progetto. A differenza di un semplice *code colocation*, dove diversi progetti sono collocati nello stesso *repository* senza relazioni definite, una *monorepo* promuove l'integrazione e la gestione coesa dei progetti. Le relazioni ben definite tra i progetti all'interno di una *monorepo* permettono una modularità e una divisione del codice che ne facilitano la manutenzione e lo sviluppo. Contrariamente alla percezione comune, una buona *monorepo* è modulare e non monolitica. La modularità permette di mantenere la separazione delle preoccupazioni e facilita il riutilizzo del codice tra i diversi progetti. Una *monorepo* ben strutturata si oppone all'idea di un *monolite*, che invece rappresenta un grande blocco di codice non suddiviso in parti discrete. Per contrasto, un *multirepo* è l'approccio standard attuale per lo sviluppo delle applicazioni, dove ogni *team*, applicazione o progetto ha il proprio *repository*. Questo metodo permette autonomia ai *team* nello scegliere le librerie, nei tempi di *deploy* delle applicazioni e su chi può contribuire al codice. Tuttavia, questa autonomia è ottenuta tramite l'isolamento, che può danneggiare la collaborazione e creare vari problemi di

gestione delle dipendenze e di integrazione. Utilizzare una *monorepo*, quindi, offre il vantaggio di una maggiore integrazione e collaborazione tra i *team*, pur mantenendo la modularità e la divisione del codice necessarie per una gestione efficace e scalabile del progetto.

Figura 6.1: Monolith vs Multirepo vs Monorepo

6.2 Gestione delle dipendenze

Una *monorepo* può avere strutture diverse e una differenza fondamentale la fa la gestione delle dipendenze. All'interno dei progetti *JavaScript*, un primo approccio per creare il proprio progetto in *monorepo* potrebbe essere utilizzando *NPM Workspaces*. Nel progetto di Stage *DDC Service*, questo è stato introdotto in questa maniera. Utilizzare *NPM Workspaces* è un approccio molto semplice che non richiede modifiche strutturali importanti alla *codebase*; tuttavia, è adatto per progetti con una complessità inferiore. Finché si hanno due applicazioni e due o tre librerie condivise, questa struttura potrebbe andare bene, ma quando si introducono gradi di complessità maggiori, come nel caso di *DDC Service*, questo approccio non è facilmente utilizzabile. Già arrivando a un grado di complessità come l'attuale struttura della *monorepo* con quattro app e cinque *packages*, la *monorepo* non è scalabile e manutenibile. La differenza la fa la gestione delle dipendenze. Con gestione delle dipendenze non si intende solo quali pacchetti esterni si ha bisogno, ma soprattutto di quale versione.

6.2.1 Gestione delle dipendenze decentralizzata

L'approccio con *NPM Workspaces* utilizza una struttura decentralizzata per la gestione delle dipendenze, delegando a ciascun progetto il compito di definire le proprie dipendenze nel proprio file `package.json`. Questo approccio decentralizzato può portare a diversi problemi. Per illustrare meglio questa struttura, possiamo fare

riferimento al seguente albero gerarchico che descrive la struttura di una *monorepo* ipotetica:

```
/
├── package.json
├── node_modules
├── apps
│   ├── A
│   │   ├── package.json
│   │   └── node_modules
│   ├── B
│   │   ├── package.json
│   │   └── node_modules
│   └── C
│       ├── package.json
│       └── node_modules
└── packages
    ├── P1
    │   ├── package.json
    │   └── node_modules
    └── P2
        ├── package.json
        └── node_modules
```

6.2.1.1 Problemi nella gestione delle dipendenze decentralizzata

Nella struttura sopra illustrata, vediamo che ogni applicazione e pacchetto ha il proprio file `package.json` per gestire le dipendenze. Questo può portare a diversi problemi:

- **Conflitti di Versione**: Diverse versioni della stessa libreria possono causare errori difficili da diagnosticare e risolvere. Per esempio, se l'applicazione A richiede `dependencyZ@1.0.0` e l'applicazione C richiede `dependencyZ@2.0.0`, gestire queste versioni diverse può essere complicato e potrebbe portare a errori durante l'esecuzione.

- **Duplicazione di Codice**: Gestire dipendenze simili in più file `package.json` può portare a duplicazione di codice e aumento delle dimensioni del repository. Ad esempio, se l'applicazione A e B hanno la stessa dipendenza esterna come `dependencyX@1.0.0`, devono dichiararla in modo ridondante in ciascun loro file `package.json`. Questo aumento della duplicazione potrebbe introdur-

re inconsistenze nel tempo e richiede un aggiornamento manuale in ciascuna dichiarazione quando si desidera effettuare un aggiornamento della dipendenza.

- **Difficoltà di Manutenzione**: Aggiornare le dipendenze in una *monorepo* decentralizzata può richiedere aggiornamenti manuali in molti file, aumentando il rischio di errori.

- **Funzionamento improprio di NPM**: Un problema comune è quando una dipendenza viene installata nel posto sbagliato. Se una dipendenza è installata nel posto sbagliato, potrebbe non essere rilevato immediatamente e le applicazioni potrebbero funzionare senza segnalare problemi. Il file `package.lock.json` di NPM memorizza le informazioni sulla *dependency tree* e le posizioni delle dipendenze all'interno della cartella `node_modules`.
  Ad esempio, se `dependencyX@1.0.0` è installata accidentalmente nell'applicazione B invece che in A, l'applicazione A potrebbe ancora accedere a `dependencyX` attraverso il percorso di B, rendendo difficile la rilevazione di errori. Consideriamo ora un caso ipotetico più complicato: se si vuole installare `dependencyY@2.0.0` come dipendenza di B, ma `dependencyY@2.0.0` richiede `dependencyX@2.0.0` come dipendenza, ci sarebbe un conflitto con `dependencyX@1.0.0` già presente in B. Sovrascrivere `dependencyX@1.0.0` con `dependencyX@2.0.0` potrebbe risolvere il problema per B, ma potrebbe rendere A incompatibile con la nuova versione di `dependencyX`.

6.2.1.2 Problema delle peer dependencies

Un problema comune nelle *monorepo* decentralizzate sono le *peer dependencies*. Le *peer dependencies* sono dipendenze che devono essere installate esternamente al modulo che le richiede, ma devono essere presenti per evitare errori di *runtime*. Consideriamo un caso dove l'applicazione A richiede `dependencyZ@1.0.0`, che a sua volta richiede `dependencyY@1.5.0` come peer dependency. Nel frattempo, l'applicazione B richiede `dependencyZ@2.0.0`, che richiede `dependencyY@2.0.0` come *peer dependency*. Se entrambe le applicazioni sono nella stessa *monorepo* decentralizzata e ciascuna gestisce le proprie dipendenze, potrebbero installare versioni incompatibili di `dependencyY`, causando errori durante l'esecuzione o comportamenti imprevisti.

Questi problemi evidenziano le difficoltà di mantenere la coerenza e la compatibilità delle dipendenze in una *monorepo* decentralizzata, spesso rendendo necessarie soluzioni più centralizzate per gestire con efficacia progetti complessi e scalabili.

6.2.1.3 Problemi nella Gestione delle Dipendenze di DDC Service

Nel momento in cui si è sviluppata Data Driven Cooking$_G$ non si sono tenute in considerazione le problematiche descritte nella gestione delle dipendenze decentralizzate. Nel momento in cui si è iniziato lo sviluppo di DDC Service$_G$, nelle prime settimane di sviluppo ci sono stati diversi problemi difficilmente individuabili. Ci sono stati alcuni momenti in cui durante l'installazione di moduli per *DDC Service* questo rendeva inutilizzabile *DDC*. Per poter portare avanti lo sviluppo queste problematiche non sono state considerate e lo sviluppo ha continuato su due *branch* separati. Nel momento in cui si è voluto fare il *merge* dei due *branch* queste problematiche sono venute di nuovo a galla. Sono stati evidenziati tutti i problemi descritti nella sezione superiore, come per esempio le dipendenze di *DDC* non fossero fin dall'inizio installate nel posto giusto. Si è visto come aumentando la complessità della *monorepo* di solo 2X questo ha completamente reso la *repository* inutilizzabile. Queste problematiche non sono solo colpa della mala gestione dei programmatori ma sono evidenziate dalle lacune tecniche che *NPM Workspaces* ha. Quando si inizia un nuovo progetto si è tentati di iniziare utilizzando questo strumento per definire una *monorepo*, tuttavia potrebbe essere una scelta sbagliata in quanto rende il progetto non estensibile in futuro.

6.2.2 Benefici della Gestione Centralizzata

Adottare una gestione centralizzata delle dipendenze offre numerosi vantaggi che risolvono i problemi tipici della gestione decentralizzata:

- **Consistenza**: Tutte le parti del progetto utilizzano le stesse versioni delle dipendenze, riducendo i conflitti. In una gestione decentralizzata, le diverse versioni delle stesse librerie possono causare errori difficili da diagnosticare e risolvere. La gestione centralizzata elimina questi conflitti mantenendo una versione uniforme delle dipendenze in tutto il progetto.

- **Semplificazione della Manutenzione**: Aggiornare una dipendenza richiede un'unica modifica, facilitando il mantenimento del progetto. Con la gestione decentralizzata, ogni progetto deve aggiornare le proprie dipendenze singolarmente, aumentando il rischio di errori e la complessità della manutenzione. La gestione centralizzata permette di aggiornare le dipendenze in un unico punto, riducendo il rischio di inconsistenze.

- **Riduzione della Duplicazione**: Con un unico file `package.json`, si evita la duplicazione di dipendenze in tutto il *repository*. La gestione decentralizzata può portare a duplicazioni di codice, aumentando le dimensioni del *repository* e complicando gli aggiornamenti. La gestione centralizzata riduce la duplicazione, rendendo il progetto più snello e facile da gestire.

6.3 Gestione delle Monorepo con Nx

Nx è uno strumento avanzato per la gestione delle *monorepo* che supporta la gestione centralizzata delle dipendenze. Utilizzando *Nx*, è possibile garantire che tutte le dipendenze siano allineate, migliorando la consistenza e semplificando la gestione del progetto. Questo strumento offre soluzioni per i problemi comuni nella gestione decentralizzata, come i conflitti di versione, la duplicazione del codice e la manutenzione complessa.

6.4 Benefici di Nx

I principali benefici di utilizzare *Nx* includono:

- **Aggiornamenti Globali delle Dipendenze**: *Nx* obbliga l'aggiornamento delle dipendenze globalmente per l'intero progetto, migliorando la coerenza. In una gestione decentralizzata, aggiornare le dipendenze in modo uniforme è difficile e può portare a conflitti. *Nx* assicura che tutte le parti del progetto utilizzino le stesse versioni delle dipendenze, eliminando questi problemi.

- **CI/CD Migliorato**: Una gestione centralizzata delle dipendenze facilita l'integrazione continua (CI) e la distribuzione continua (CD), riducendo i rischi

di errore. Nella gestione decentralizzata, la CI/CD$_G$ può essere compromessa da dipendenze incoerenti tra i progetti. *Nx* garantisce che tutte le dipendenze siano coerenti, migliorando la stabilità e l'affidabilità dei processi *CI/CD*.

- **Strumenti di Sviluppo Avanzati**: *Nx* offre strumenti potenti per la gestione delle *build*, dei test e delle dipendenze, migliorando l'efficienza dello sviluppo. Questi strumenti aiutano a gestire la complessità delle *monorepo*, facilitando la *build* e il *testing* delle diverse parti del progetto. Nella gestione decentralizzata, questi processi possono essere disorganizzati e inefficienti. *Nx* centralizza e ottimizza questi processi, migliorando la produttività del team di sviluppo.

6.5 Necessità di Test Approfonditi

In una monorepo con gestione centralizzata delle dipendenze, è essenziale implementare test di unità, di integrazione e E2E per garantire che ogni modifica sostanziale non comprometta l'intero progetto. Dato che le dipendenze sono condivise tra tutti i progetti, una modifica in una dipendenza può avere effetti a catena su tutto il sistema, rendendo i test critici per mantenere la stabilità e la funzionalità della monorepo. Questo aiuterebbe nelle *CI/CD* ma è necessario utilizzare test di unità o di integrazione. I classici problemi che possono verificarsi durante l'installazione o l'aggiornamento di una dipendenza sono quando un altro progetto diventa non funzionante. Per proteggersi da questo, si dovrebbe almeno testare che con ciascuna modifica gli altri progetti continuino a compilare il loro codice e ad eseguire correttamente.

Capitolo 7

Conclusioni

7.1 Consuntivo finale

7.2 Raggiungimento degli obiettivi

7.3 Conoscenze acquisite

7.4 Valutazione personale

Bibliografia

Sitografia

Dripsy. URL: https://www.dripsy.xyz (cit. a p. 14).

Integrating Graphql with Redux toolkit and RTK-query. URL: https://lashanfaliq.medium.com/integrating-graphql-with-redux-toolkit-and-rtk-query-cc8040f92bd.

Migrating to a Monorepo Using NPM 7 Workspaces. URL: https://medium.com/edgybees-blog/how-to-move-from-an-existing-repository-to-a-monorepo-using-npm-7-workspaces-27012a100269 (cit. alle pp. 33, 35).

Moti. URL: https://moti.fyi/starter (cit. a p. 13).

Peer Dependencies. URL: https://blog.bitsrc.io/understanding-peer-dependencies-in-javascript-dbdb4ab5a7be.

React Native Cookie Authentication. URL: https://javascript.plainenglish.io/react-native-cookie-authentication-83ef6e84ba70 (cit. a p. 42).

Solito. URL: https://solito.dev (cit. a p. 13).

Unox. URL: https://www.unox.com/it_it/ (cit. a p. 3).