Università degli Studi di Padova

Dipartimento di Matematica “Tullio Levi-Civita”

Corso di Laurea in Informatica

AlphaNotary: dApp per notarizzazione e monitoraggio di NFT tramite Blockchain Ethereum compatibili

Tesi di laurea

Relatore

Prof. Francesco Ranzato

Laureando

Adnan Latif Gazi

Anno Accademico 2021-2022

AlphaNotary: dApp per notarizzazione e monitoraggio
di NFT tramite Blockchain Ethereum compatibili

Adnan Latif Gazi

Ringraziamenti

Vorrei innanzitutto ringraziare il Prof. Francesco Ranzato per il supporto fornitomi durante il periodo di stage e la stesura di questa tesi. Senza il suo aiuto non sarei riuscito a svolgere tali attività correttamente.

Sono poi davvero riconoscente all'azienda Venicecom S.r.l. per la positiva esperienza di tirocinio e per i bei momenti trascorsi insieme che mi hanno fatto sentire fin da subito incluso nella vostra realtà. Ho avuto l'opportunità di confrontarmi con persone stupende da cui ho imparato tanto della vita e acquisito molte più competenze tecniche di quante mi aspettassi. Grazie alle vostre indicazioni ho potuto completare con massimo successo il progetto di lavoro.

Un ringraziamento speciale va al mio amico Marco Andrea Limongelli per essere stato una delle mie fonti di motivazione principali in questi ultimi anni. Grazie alla tua amicizia e al tuo esempio ho acquisito l'etica di pensare in grande e puntare sempre più in alto, spingendo entrambi a migliorare sempre. Sono convinto che con le tue formidabili capacità raggiungerai tutti i tuoi obiettivi. Quindi, posso solo che augurarti ogni felicità nella vita.

Esprimo inoltre la mia più sincera gratitudine a tutti i miei amici per avermi accompagnato sia nei momenti belli che in quelli brutti, grazie a cui sono sempre riuscito ad apprezzare ogni circostanza e trarne un importante insegnamento.

Ringrazio di cuore anche la mia famiglia e i parenti per tutti i sacrifici compiuti al fine di permettermi di raggiungere i miei obiettivi. Senza il vostro appoggio non sarei riuscito ad arrivare fino a questo punto. Vi prometto che non smetterò mai di rendervi orgogliosi di me.

Sono grato, infine, all'Università, ai miei colleghi studenti e a tutti i professori per questa meravigliosa esperienza grazie a cui sono assai maturato come persona.

Padova, Settembre 2022 Adnan Latif Gazi

Sommario

Il documento descrive il lavoro svolto durante lo stage presso la sede di Venezia dell'azienda *Venicecom S.r.l.* L'obiettivo del tirocinio era lo sviluppo di un'applicazione in grado di autenticare documenti grazie all'uso delle *Blockchain*, più dettagliatamente espresso come *AlphaNotary*: una *dApp* per la *notarizzazione* e il monitoraggio di *NFT* tramite *Blockchain Ethereum* compatibili.

In particolare, di seguito vengono in ordine esposti il contesto lavorativo in cui si è attuato il periodo di tirocinio, le caratteristiche concordate per lo svolgimento dello stage e lo sviluppo del progetto, le sue fasi di progettazione e realizzazione, l'analisi del prodotto finale e le conclusioni generali sull'esperienza.

Infine, si rammenta che le seguenti convenzioni sono state adottate:

- Nomi di tecnologie e parole tecniche sono in corsivo.
- Figure relative a codice sono state inserite nell'appendice alla fine del corpo del documento e referenziate man mano utilizzando la propria didascalia.
- Acronimi, abbreviazioni, parole ambigue, meno comuni e più caratteristiche del progetto sono descritte approfonditamente nel glossario in fondo al documento.
- Tutte le fonti esterne vengono elencate alla fine del documento e citate tra parentesi quadre con il loro simbolo lì associato man mano che sono state utilizzate.

Indice

1 Contesto lavorativo

1.1 Presentazione dell'azienda

Figura 1.1: Logo Venicecom S.r.l.

Venicecom S.r.l. è una *start-up* nata a Venezia nel 1997 per colmare le crescenti richieste in ambito *Web Agency* e *Application Service Provider*, e presto allargatosi nel ramo *Software House* e *System Integrator* [S1]. L'azienda è subito diventata rilevante nel settore originario e, grazie alla costante espansione riscontrata fin dai primi anni, si è trasformata lentamente in un gruppo internazionale di consulenza e servizi *IT*, con sede principale a Venezia, e località operative a Bari, Roma, due in Kazakhstan, Romania e Mozambico, in cui ognuna vanta in media circa trenta dipendenti e forma gruppi eterogenei e uniformi che affianca personale di ogni età, esperienza, ruolo e settore, in modo da garantire continua crescita dei dipendenti sotto ogni aspetto tecnico e personale [S2].

L'impresa si è distinta grazie ai suoi tre seguenti software utilizzati da partner di rilievo, quali Eni, Generali, AVM, Eurospin, Nato, regione Abruzzo, Città Metropolitana di Venezia, città di Roma e molte altre aziende private ed enti pubblici:

- ***ProQ***: piattaforma di *e-Procurement*, che quindi permette di gestire le gare d'appalto automatizzando le procedure d'acquisto, frutto di più della metà dei guadagni totali [S3].
- ***Kalmo***: software suite per la gestione e regolamentazione della salute, sicurezza e formazione del personale in contesti di lavoro a rischio elevato [S4].
- ***KronosApp***: sistema di reportistica per la gestione, pianificazione e controllo delle commesse allo scopo di ridurre i rischi e i costi, aumentando al tempo stesso la redditività [S5].

Questi prodotti, che forniscono la quasi totalità del fatturato della società, richiedono costante manutenzione, assistenza ai clienti e sono sempre soggette a evoluzioni, rendendo l'azienda una *Software House* orientato anche alla consulenza. Sono stati inoltre mantenuti i servizi *IT* iniziali, quali quelli di *Web Agency, Application Service Provider* e *System Integrator*, che però producono ormai solo una parte complementare dei ricavi.

Questa piccola multinazionale, nonostante la rilevanza, le dimensioni e le ambizioni di crescita continua, mantiene da sempre gli stessi valori alla base del proprio successo: semplicità del lavoro, spirito di servizio, responsabilità individuale, ricerca dell'eccellenza e accentramento delle persone, sia clienti che colleghi, come priorità principale [S6].

1.2 Modello d'impresa

1.2.1 Processo di lavoro

Lo sviluppo di progetti in azienda sfrutta il modello, ovvero il concetto organizzativo di un lavoro, *Agile*, che garantisce una distribuzione continua di software e loro funzionalità in modo rapido e iterativo, un ampio controllo del progetto che rende possibile concepire soluzioni migliori e più rapidamente, una maggior flessibilità e leggerezza della struttura lavorativa a vantaggio della produttività e una riduzione del rischio di risultati fallimentari con conseguente aumento della qualità del prodotto finale [S7].

In particolare, viene sfruttata la metodologia, ovvero la struttura organizzativa di un lavoro ricavata applicando un modello, *Scrum*, ottenuto da quello *Agile*, che permette una comunicazione di qualità, cioè efficace ed efficiente, nel gruppo e con il cliente, di essere sempre disponibili a migliorare il prodotto in base alle richieste di quest'ultimo, una comprensione immediata e chiara del lavoro da entrambe le parti e una definizione più semplice e precisa dei requisiti [S8].

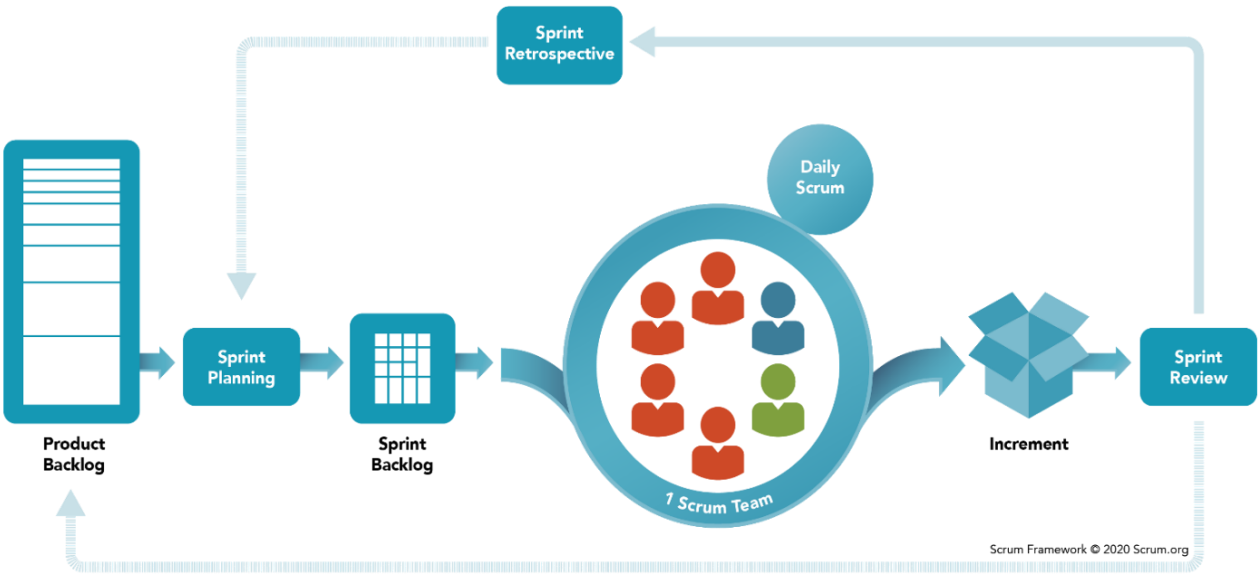

Figura 1.2: Metodologia Scrum

Un progetto viene affidato ad uno *Scrum team*, ovvero un gruppo di sviluppo *software* di piccole dimensioni, al massimo dieci membri, che include le seguenti figure presenti in numero adeguato in base al lavoro da svolgere:

- **Responsabile**: rappresenta il progetto, conosce le richieste del cliente, con cui dialoga frequentemente e ne fornisce indicazioni alla squadra, e ricopre il ruolo di *Scrum master*, che aiuta il gruppo a rispettare i propri obiettivi e a rimuovere gli ostacoli alla produttività.
- **Amministratore/i**: redige i *backlog* e ricopre il ruolo del *project manager*, che gestisce la metodologia di lavoro e amministra le attività.

- **Analista/i:** raccoglie le indicazioni del cliente, analizza il progetto, classificando e mantenendo aggiornato i requisiti del prodotto finale e la loro fattibilità, e collabora al processo di progettazione.
- **Progettista/i:** definisce una soluzione al progetto che rispetti i requisiti ottenuti dalle analisi, assiste e fornisce indicazioni al processo di sviluppo e da queste prende spunto per mantenere aggiornata la soluzione.
- **Programmatore/i:** sviluppa la soluzione al progetto seguendo le indicazioni del progettista e si alterna con il verificatore.
- **Verificatore/i:** controlla la correttezza dello sviluppo e si alterna con il programmatore.

Ogni progetto prevede l'uso dei propri *backlog*, ovvero documenti tecnici e sintetici, che quindi assumono la forma di liste, sempre soggette a miglioramenti e in cui vengono inserite tutte le note stabilite nel corso del lavoro. Questi sono:

- ***Kanban*:** tabellone costantemente aggiornato che rappresenta le attività in base al loro stato di avanzamento, in modo da visualizzare la situazione del lavoro, sia globale che specifica, e aiutare la stipula del piano di progetto e di qualifica.
- **Piano di progetto**: contiene tutte le informazioni su come la metodologia di lavoro è stata applicata nel tempo, tracciando quindi le fasi, gli sprint, gli obiettivi, gli stati delle attività, le ore e i costi, in modo da poter controllare il rispetto del modello d'impresa.
- **Piano di qualifica**: classifica approfonditamente tutti i dettagli dello sviluppo del prodotto in modo da calcolarne l'indice di realizzazione nel tempo, utile a monitorare l'avanzamento del risultato finale e accertare l'efficacia del lavoro.
- **Norme di progetto:** raccoglie tutte le regole e convenzioni adottate dal gruppo, descrivendole e motivandole dettagliatamente, facilitando l'avvio al lavoro di nuovi componenti, evitando ambiguità nell'esecuzione delle attività e aumentando la redditività.

Il lavoro viene suddiviso equamente in *sprint*, intervalli temporali la cui durata è arbitraria e dipende dal progetto, dall'azienda, dal gruppo di sviluppo e dai successivi obiettivi da raggiungere, ma che solitamente è di circa due settimane. Ogni *sprint* deve raggiungere almeno l'obiettivo più vicino, dove ognuno rappresenta un incremento del prodotto e del suo valore tramite l'implementazione di funzionalità utilizzabili. Più *sprint* permettono di raggiungere una *milestone*, cioè un punto chiave dello sviluppo del progetto ottenuto dalla concretizzazione di molteplici obiettivi.

Gli sprint sono preceduti da una riunione di gruppo per la pianificazione degli obiettivi. Ogni *sprint* termina con uno *sprint review*, ovvero una riunione tra la quadra e il cliente con l'obiettivo di mostrare i risultati dell'ultimo periodo attraverso dimostrazioni del funzionamento di quanto appena ottenuto e il relativo prodotto finale. Segue anche un'assemblea di gruppo per effettuare una retrospettiva del periodo appena concluso, analizzandone i pro e contro in tutti i suoi passi, in modo da definire piani su come migliorare la qualità degli *sprint* successivi. Inoltre, all'inizio di ogni giornata lavorativa, il gruppo si riunisce in uno *stand-up meeting*, ovvero una riunione in media di circa 15 minuti, tenutasi in piedi appunto da essere breve, per discutere dell'andamento del lavoro.

Le riunioni vengono amministrate, per quanto riguarda la rappresentanza della squadra e il dialogo con l'eventuale cliente, dal responsabile, che, in veste anche del ruolo di *Scrum master*, ne gestisce le tempistiche, le modalità e l'ordine. Le riunioni presentano la seguente struttura della discussione:

- Andamento del lavoro dalla precedente riunione.
- Eventuali problemi sorti dalla passata riunione.
- Attività e obiettivi totali e di ogni membro fino alla riunione successiva, scritti in linguaggio naturale sottoforma di casi d'uso che rappresentano le funzionalità da implementare.
- Indicazioni utili alla risoluzione dei problemi e materiali validi alla formazione di ogni componente per lo svolgimento del proprio lavoro corrente.

1.2.2 Processo di distribuzione

Per via del processo di lavoro usato nell'azienda, periodicamente occorre integrare prodotti in quel momento già in esecuzione e anche soggetti a sviluppo con funzionalità appena realizzate: a tal proposito si usufruisce del metodo di distribuzione *software Continuous Integration/Continuous Delivery* (*CI/CD*). In particolare, *Continuous Integration* (*CI*) permette di accrescere il codice sorgente di un sistema di componenti nuove, garantendo la loro correttezza e *versionando* il prodotto finale, mentre *Continuous Delivery* (*CD*) consente di aggiungere tali funzionalità al programma già in esecuzione senza mai interrompere il servizio.

Questo metodo apporta anche molteplici vantaggi, quali riduzione del tempo di approvazione e distribuzione attraverso le automatizzazioni fornite, scoperta e correzione di errori più velocemente e miglioramento della qualità del prodotto grazie ai molteplici stadi di verifica, monitoraggio del ciclo di vita delle componenti appena introdotte e ricavo delle prestazioni attraverso gli strumenti di controllo offerto dai servizi del metodo [S9].

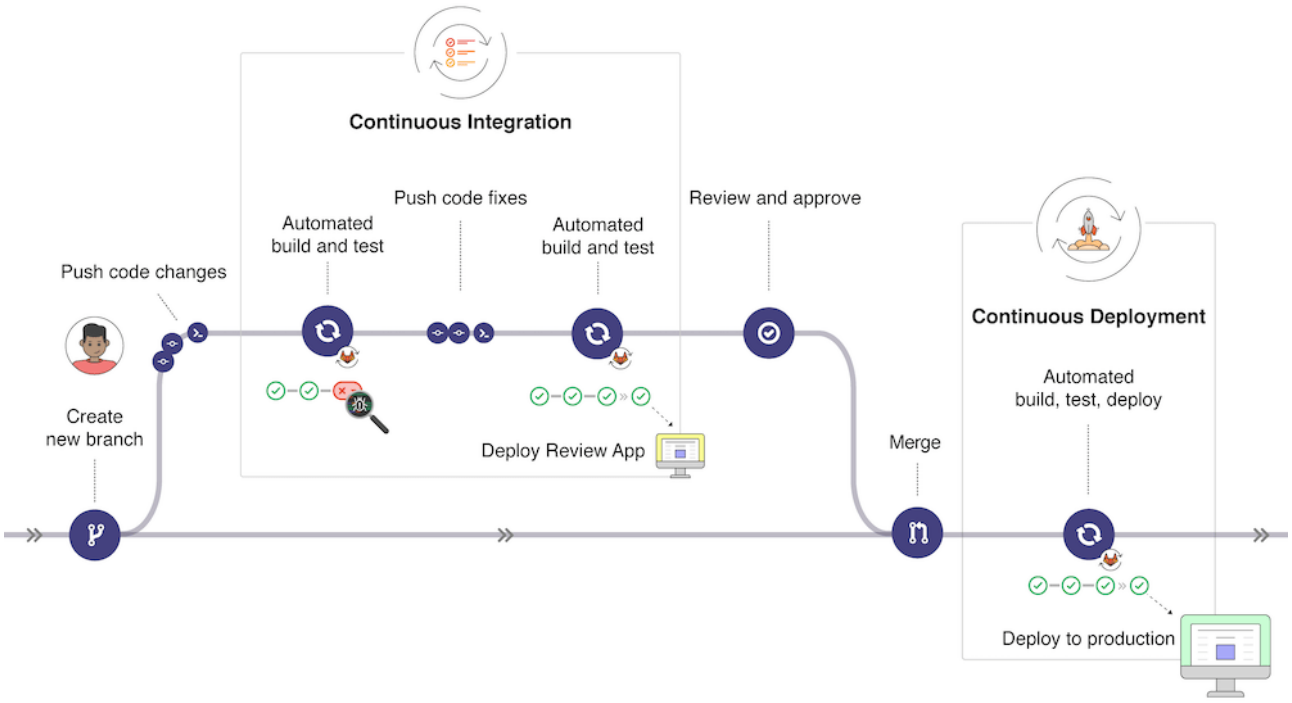

Figura 1.3: Continuous Integration/Continuous Delivery (CI/CD)

Il processo di distribuzione si svolge eseguendo le seguenti fasi:

A. ***Continuous Integration***:
   1. Realizzazione e man mano verifica locale.
   2. Superamento di *test* automatici per la verifica non locale.
   3. Revisione del codice per la validazione.

B. ***Continuous Delivery*****:**

1. Rilascio automatico nel *repository*.
2. Distribuzione automatica in produzione.

In cui bisogna tenere in considerazione che:

- Il mancato superamento di verifiche locali, *test* automatici o revisioni, comporta il ritorno alla fase di realizzazione.
- La verifica locale assicura che il risultato funzioni almeno in qualche situazione, i *test* automatici che funzioni in ogni circostanza, mentre la revisione assicura la qualità del codice.
- I *test* automatici vengono preventivamente costruiti da altri programmatori dopo aver ricevuto i *pre* e i *post* condizioni delle funzionalità da sviluppare.
- Il rilascio automatico nel *repository* necessita di risoluzione manuale dei conflitti.
- La distribuzione automatica in produzione sfrutta un sistema di *pipeline*, anch'esso automatizzato, per non interrompere mai il servizio.

Tutti i progetti devono essere condivisi su domini aziendali: quelli soggetti a questo metodo di distribuzione prevedono il loro caricamento su *repository* appartenenti all'impresa, quelli che non sono *software* devono essere invece condivisi sul *cloud* dell'impresa per avere un *backup,* e tutti gli altri devono essere caricati e avviati sul *server* aziendale per avere il servizio disponibile all'uso all'interno dell'azienda.

2 Descrizione dello stage

2.1 Presentazione del progetto

2.1.1 Dominio applicativo del problema

ProQ richiede per motivi burocratici la stipula ad ogni procedura d'acquisto di molteplici documenti in formato digitale sia all'azienda che ai clienti. Assume quindi un'importanza rilevante la gestione di questi certificati, a partire dalla garanzia d'autenticità e la possibilità di condivisione, fino alla sicurezza di conservazione e i costi esecutivi per l'uso.

L'azienda usufruisce a tal proposito di un'applicazione in grado di amministrare gli attestati in modo congruo alle suddette necessità. Dopo aver compilato un nuovo atto, esso dev'essere caricato all'interno di questo sistema, che ne ricava e memorizza una chiave. Quando invece si vuole verificare l'autenticità di un certificato, questo dev'essere nuovamente aggiunto all'applicazione, così da potervi calcolare una seconda volta la chiave per controllarne la presenza. Se il contratto non avrà subito modifiche nel tempo, la chiave appena prodotta sarà uguale a quella salvata originariamente, quindi l'applicazione sarà in grado di trovarla e fornire esito positivo. Se invece il documento sarà stato modificato precedentemente, verrà prodotta una chiave diversa da quella iniziale, facendo fallire la ricerca e avvertendo l'utente che il certificato in suo possesso o è stato cambiato da quello originale, oppure che si tratta semplicemente di un nuovo attestato, e quindi non ancora presente nel sistema.

Le chiavi vengono salvate in *Blockchain* in modo da garantire la sicurezza dei contratti e ridurre notevolmente i costi di mantenimento, che però, sono comunque molto elevati in quanto l'applicazione non è proprietaria dell'azienda. Oltretutto, essendo il sistema interno all'impresa, nel senso che è presente solamente l'eseguibile del programma avviato sul server aziendale, viene meno anche la capacità di condivisione degli atti al di fuori della società, ovvero a chiunque non sia in grado di accedere alla rete aziendale, e quindi collegarsi al suo server.

2.1.2 Sistema da sviluppare

Si vuole creare un'applicazione, nominata successivamente *AlphaNotary*, analoga a quella già in utilizzo dall'azienda, ma di essa proprietaria, in modo da eliminare le dipendenze dai sistemi esterni, riducendo così i costi di mantenimento e semplificando la gestione della sua infrastruttura. Per adempiere all'esigenza di condivisione dei documenti, il nuovo sistema dovrà essere una *dApp*, cioè un'applicazione decentralizzata in grado di operare autonomamente, con possibilità di essere usufruito da tutti e che usufruiscono di *Blockchain*. Per permettere invece maggiore versatilità e aumentare così gli scenari d'uso del servizio, sarebbe gradito se il sistema potesse gestire ogni genere di file, espandendosi in questo modo all'amministrazione degli *NFT*, ovvero un elemento digitale unico e non riproducibile.

Si desidera inoltre arricchire l'applicazione con un'interfaccia di monitoraggio in grado di visualizzare i risultati delle operazioni del sistema al fine di accertare il suo corretto funzionamento. Essa dovrà anche essere modulare, ovvero con possibilità di collegarlo e separarlo dall'applicazione a piacimento in modo semplice e senza intaccare il restante servizio.

Infine, si aspira ad espandere l'infrastruttura di una serie di funzionalità complementari per assicurare una gestione completa degli attestati coinvolti nel sistema e le informazioni utente necessarie all'uso del servizio. Sarebbe apprezzato se anche la gestione delle *Blockchain* potesse essere altrettanto modulare e completa.

Il sistema sarà quindi per la gestione di documenti su *Blockchain*. In particolare, dovrà principalmente eseguire *notarizzazione*, ovvero autenticazione, e monitoraggio di *NFT* tramite tutte le *Blockchain Ethereum* compatibili, in quanto le uniche che consentono di sviluppare tali servizi.

2.2 Vincoli da soddisfare

2.2.1 Requisiti

Dopo un attento studio di fattibilità coinvolgente lo studente e il tutor aziendale, in veste anche da project manager del progetto, in cui sono state analizzate le tempistiche a disposizione, la capacità dello studente, le richieste del progetto e tutti gli altri vincoli organizzativi imposti, è stato concordato lo sviluppo delle seguenti componenti, suddivisi per priorità e ordine di svolgimento:

- **Obbligatorie:** requisiti primari, necessari alla buona riuscita dello stage.
    1. ***DApp* di *notarizzazione*:** realizzazione di una *dApp* per la *notarizzazione* e il monitoraggio di *NFT* tramite *Blockchain Ethereum* compatibili.
    2. **Interfaccia di monitoraggio:** realizzazione di un'interfaccia di monitoraggio in grado di visualizzare i risultati delle operazioni della *dApp* di *notarizzazione* al fine di accertare il suo corretto funzionamento.
- **Desiderabili:** non necessari, ma che contribuiscono alla completezza del prodotto.
    1. **Documentazione della *dApp* di *notarizzazione*:** stesura della documentazione che ha il compito di presentare la *dApp* di *notarizzazione* e di descriverne il processo di realizzazione.
    2. **Documentazione dell'interfaccia di monitoraggio:** stesura della documentazione che ha il compito di presentare l'interfaccia di monitoraggio e di descriverne il processo di realizzazione.
- **Opzionali:** puramente superflui, che portano valore aggiunto al progetto.
    1. **Funzionalità complementari:** realizzazione di una serie di funzionalità per assicurare una gestione completa degli attestati coinvolti nella *dApp* di *notarizzazione* e le informazioni utente necessarie all'utilizzo del servizio.
    2. **Documentazione delle funzionalità complementari:** stesura della documentazione che ha il compito di presentare le funzionalità complementari e di descriverne il processo di realizzazione.

Si rammenta che la documentazione implica anche progettazione, mentre la realizzazione include collaudo e distribuzione. In particolare, sono stati previsti i seguenti caricamenti dei prodotti finali:

- Man mano il codice sorgente in un *repository* condiviso e soggetto a *Continuous Integration/Continuous Delivery* (*CI/CD*) per consentire futuro mantenimento e sviluppi.
- A fine tirocinio tutto il lavoro sul *cloud* dell'impresa per avere un *backup*.
- A fine tirocinio l'seguibile del programma sul *server* aziendale e avviato per avere il servizio disponibile all'uso all'interno dell'azienda.

Essi rappresentano i requisiti del progetto, necessari a tirocinio concluso per quantificare il grado di successo del lavoro, e quindi per caratterizzare la valutazione. In particolare, è stato deciso che, per avere un prodotto usabile, le componenti obbligatorie dovessero essere necessariamente completate, mentre le altre sarebbero state sviluppate dopo e per quanto possibile nel tempo rimanente. In ogni caso, la quantità di lavoro è stata calibrata per riuscire a terminare tutte le attività entro lo stage.

2.2.2 Obiettivi

L'unico obiettivo obbligatorio dell'azienda è formare inizialmente lo studente sui domini coinvolti nel progetto, in modo da non precludere la possibilità di future collaborazioni. Più precisamente, l'azienda vuole proseguire lo sviluppo del sistema ed espandere l'uso delle *Blockchain* in molteplici contesti d'uso. La redazione dei documenti servirà successivamente per l'espansione di questo prodotto e facilitare la comprensione, e quindi la realizzazione, di quelli analoghi. Infine, è ovviamente scopo dello stage introdurre lo studente nel mondo del lavoro.

2.3 Metodologia di lavoro

2.3.1 Processo di sviluppo

Gli orari di lavoro andavano dal lunedì al venerdì dalle 9:00 alle 18:00, con un'ora di pausa pranzo dalle 13:00 alle 14:00. Veniva comunque concessa una discreta flessibilità su di esse permettendo di lavorare in qualunque orario rimanendo nell'arco temporale di apertura dell'azienda, ovvero dalle 8:30 alle 18:30.

È stato deciso di svolgere il lavoro in autonomia, usufruendo però del modello d'impresa dell'azienda, in cui il tutor aziendale ha rivestito la parte del responsabile e del cliente, lo studente ha ricoperto il ruolo del resto del gruppo e i *backlog* sono la documentazione requisito del progetto. In ogni caso, è stato comunque garantito da parte del tutor aziendale un'ampia disponibilità per ulteriori incontri, se richiesti da una delle due parti, e una costante assistenza, soprattutto nelle fasi di formazione dello stage, ovvero quelle iniziali.

Le interazioni sono state svolte di persona o in remoto in base alla situazione e modalità di lavoro di allora. Alle interazioni hanno partecipato anche altre persone utili allo sviluppo del progetto o al coordinamento delle attività. La modalità di lavoro era in presenza, con possibilità di lavorare da remoto per necessità dello studente o dell'azienda, o per l'assenza del tutor aziendale: in questo caso lo studente ha avuto scelta di quale modalità usufruire. Le interazioni da remoto sono state attuate tramite e-mail, telefono o video chiamate. Sono state infine adoperate tutte le modalità aziendali relative ai processi di distribuzione.

2.3.2 *Software* e strumenti

I seguenti *software* e strumenti sono stati usati per la gestione del lavoro:

- ***Visual Studio Code*:** ambiente di sviluppo di codice sorgente. Usato per la realizzazione del codice sorgente del programma.

Figura 2.1: Logo Visual Studio Code

- ***Git*:** *software* di controllo di versione distribuito di pacchetti di codice sorgente. Usato per il *versionamento* delle librerie.

Figura 2.2: Logo Git

- ***GitHub*:** servizio di hosting con implementazione degli strumenti di *Git*. Usato per la conservazione, condivisione e *versionamento* del programma.

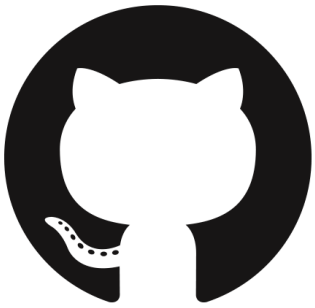

Figura 2.3: Logo GitHub

- ***Putty*****:** *client SSH* per la gestione da remoto di sistemi informatici. Usato per la connessione al *server* aziendale.

Figura 2.4: Logo Putty

- ***Microsoft Teams*****:** piattaforma di comunicazione orientato alla collaborazione lavorativa, usato per l'interazione informale con i colleghi.

Figura 2.5: Logo Teams

- ***Outlook*****:** servizio di posta elettronica. Usato per le comunicazioni formali, condivisioni di file e organizzazione delle riunioni.

Figura 2.6: Logo Outlook

- ***Microsoft Word*****:** programma di videoscrittura. Usato per la stesura della documentazione.

Figura 2.7: Logo Microsoft Word

- ***Microsoft Excel*:** programma di elaborazione di fogli elettronici. Usato per la produzione tabelle di calcolo relative allo stage e al programma.

Figura 2.8: Logo Excel

- ***Dropbox*:** servizio di *file hosting*. Usato per la condivisione di tutto il lavoro.

Figura 2.9: Logo Dropbox

- ***Vercel*:** piattaforma di distribuzione di applicazioni *web*. Usato per la distribuzione continua del prodotto.

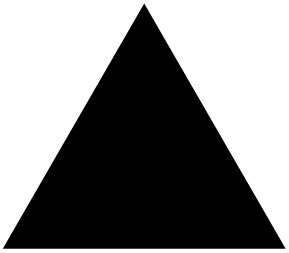

Figura 2.10: Logo Vercel

- ***Apache HTTP Server*:** piattaforma *server web Linux* tramite protocollo *HTTP*. Usato per l'installazione e l'avvio dell'eseguibile del programma sul *server* aziendale.

Figura 2.11: Logo Apache HTTP Server

- ***Microsoft Windows*:** sistema operativo principale per lo svolgimento del progetto.

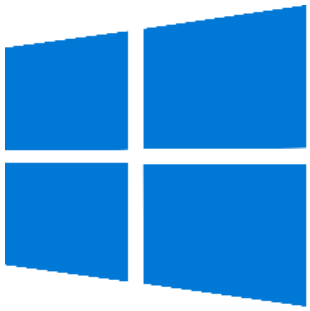

Figura 2.12: Microsoft Widows

- ***Linux*:** sistema operativo secondario per la verifica del corretto funzionamento del programma anche in diversi sistemi operativi e per lavorare sul *server* aziendale.

Figura 2.13: Logo Linux

2.4 Organizzazione del lavoro

2.4.1 Fasi di lavoro

Suddividendo in modo uniforme il lavoro previsto per completare ogni requisito e unendolo con gli obiettivi, sono state ottenute le seguenti fasi di lavoro:

1. Introduzione e formazione sui domini coinvolti nel progetto.
2. Realizzazione del *back-end* della *dApp* di *notarizzazione*.
3. Documentazione del *back-end* della *dApp* di *notarizzazione*.
4. Realizzazione del *front-end* della *dApp* di *notarizzazione*.
5. Documentazione del *front-end* della *dApp* di *notarizzazione*.
6. Realizzazione dell'interfaccia di monitoraggio.
7. Documentazione dell'interfaccia di monitoraggio.
8. Realizzazione delle funzionalità complementari.
9. Documentazione delle funzionalità complementari.

Essendo che ogni fase di lavoro porta alla realizzazione di una funzionalità da aggiungere al prodotto finale, nel modello d'impresa adottato, questi corrispondono a obiettivi attorno a cui organizzare gli *sprint*, settimanali in questo caso in quanto le fasi di lavoro sono pari al numero di settimane di tirocinio, definendo quindi una pianificazione generica delle attività.

2.4.2 Dettaglio delle attività

Lo stage prevedeva 8 settimane di lavoro a tempo pieno, per un totale di 320 ore, collocate in un periodo di 9 settimane reali, dal 23 maggio 2022 al 22 luglio 2022, in modo da lasciare margine di sicurezza per ogni evenienza che costringesse ad allungare il tirocinio. La pianificazione a dettaglio delle attività, ottenuto approfondendo le fasi di lavoro, è stata così definita:

- **Prima Settimana:** introduzione e formazione sui domini coinvolti nel progetto (35 ore).
    1. Presa visione dell'infrastruttura aziendale (5 ore).
    2. Configurazione dell'ambiente di lavoro discussione al dettaglio del progetto (10 ore).
    3. Formazione sui domini coinvolti nel progetto: *Blockchain*, *dApp*, *notarizzazione*, *NFT*, *Blockchain Ethereum* compatibili (20 ore).
- **Seconda settimana:** realizzazione del *back-end* della *dApp* di *notarizzazione* (35 ore).
    1. Studio del *back-end* della *dApp* di *notarizzazione* (7.5 ore).
    2. Progettazione del back-*end* della *dApp* di *notarizzazione* (7.5 ore).
    3. Realizzazione del *back-end* della *dApp* di *notarizzazione* (20 ore).
- **Terza settimana:** documentazione del *back-end* della *dApp* di *notarizzazione* (35 ore).
    1. sviluppo e distribuzione del *back-end* della *dApp* di *notarizzazione* (20 ore).
    2. Verifica, collaudo e distribuzione del *back-end* della *dApp* di *notarizzazione* (7.5 ore).
    3. Documentazione del *back-end* della *dApp* di *notarizzazione* (7.5 ore).
- **Quarta settimana:** realizzazione del *front-end* della *dApp* di *notarizzazione* (35 ore).
    1. Studio del *front-end* della *dApp* di *notarizzazione* (7.5 ore).
    2. Progettazione del *front-end* della *dApp* di *notarizzazione* (7.5 ore).
    3. Realizzazione del *front-end* della *dApp* di *notarizzazione* (20 ore).
- **Quinta settimana:** documentazione del *front-end* della *dApp* di *notarizzazione* (35 ore).
    1. Sviluppo del *front-end* della *dApp* di *notarizzazione* (20 ore).
    2. Verifica, collaudo e distribuzione del *front-end* della *dApp* di *notarizzazione* (7.5 ore).
    3. Documentazione del *front-end* della *dApp* di *notarizzazione* (7.5 ore).
- **Sesta settimana:** realizzazione dell'interfaccia di monitoraggio (35 ore).
    1. Studio dell'interfaccia di monitoraggio (7.5 ore).
    2. Progettazione dell'interfaccia di monitoraggio (7.5 ore).
    3. Realizzazione dell'interfaccia di monitoraggio (20 ore).
- **Settima settimana:** documentazione dell'interfaccia di monitoraggio (35 ore).
    1. Sviluppo dell'interfaccia di monitoraggio (20 ore).
    2. Verifica, collaudo e distribuzione dell'interfaccia di monitoraggio (7.5 ore).
    3. Documentazione dell'interfaccia di monitoraggio (7.5 ore).
- **Ottava settimana:** realizzazione delle funzionalità complementari (35 ore).
    1. Studio delle funzionalità complementari (7.5 ore).
    2. Progettazione delle funzionalità complementari (7.5 ore).
    3. Realizzazione delle funzionalità complementari (20 ore).
- **Nona settimana:** documentazione delle funzionalità complementari (40 ore).
    1. Sviluppo delle funzionalità complementari (20 ore).
    2. Verifica, collaudo e distribuzione delle funzionalità complementari (7.5 ore).
    3. Documentazione delle funzionalità complementari (7.5 ore).
    4. Dimostrazione e consegna del prodotto finale (5 ore).

2.4.3 Pianificazione di processi e fasi

La pianificazione, calcolata sommando le ore di lavoro dal dettaglio delle attività relative ai seguenti processi, è stata così ripartita a livello di ore di lavoro per processo:

Durata in ore	Descrizione dell'attività
65	**Formazione sui domini coinvolti nello sviluppo del sistema**
65	**Progettazione e documentazione**
10	*Analisi del problema e del dominio applicativo*
20	*Progettazione della soluzione e relativi test*
5	*Pianificazione del lavoro*
30	*Documentazione sulla progettazione, realizzazione e presentazione*
150	**Realizzazione dei sistemi e implementazione delle funzionalità**
75	*Realizzazione dei sistemi*
75	*Implementazione delle funzionalità*
40	**Verifica, collaudo e dimostrazione**
25	*Verifica della progettazione e del codice*
10	*Collaudo dei sistemi realizzati e delle funzionalità implementate*
5	*Dimostrazione del sistema finale agli stakeholders*
Totale ore	**320**

Figura 2.14: Preventivo tabella ripartizione delle ore di lavoro per processo

La pianificazione, calcolata invece dalle ore di lavoro dal dettaglio delle attività relative ad ogni fase di lavoro, è stata ripartita a livello temporale per fase di lavoro come mostrato dal seguente diagramma di *Gantt*:

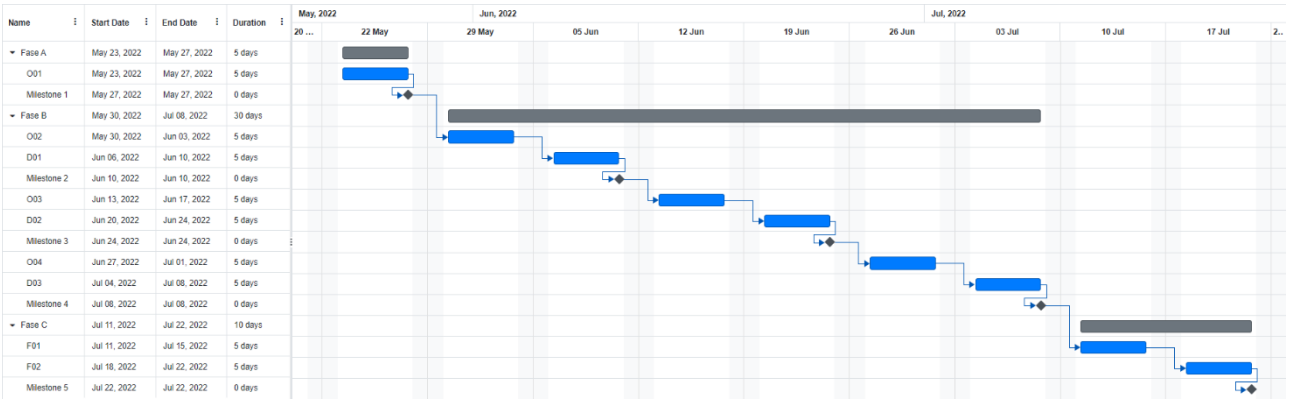

Figura 2.15: Preventivo diagramma di Gantt ripartizione del tempo per le fasi di lavoro

3 Progettazione

3.1 Infrastruttura esterna

3.1.1 Interazioni esterne

L'infrastruttura esterna, cioè l'apparato di sistemi su cui l'applicazione si basa per il suo funzionamento, è molto complessa e vale sicuramente la pena di essere analizzata in modo da facilitare la comprensione dell'infrastruttura interna, ovvero l'architettura del sistema. L'infrastruttura esterna è composta dai seguenti sistemi che comunicano tra loro come di seguito descritto per permettere all'applicazione di usufruire delle *Blockchain*:

- ***Blockchain*:** struttura dati a forma di lista concatenata in cui sono ammessi solamente operazioni di lettura e scrittura. In quest'ultima, i dati vengono *crittografati*, raggruppati equamente e inseriti in pacchetti chiamati blocchi, a loro volta *crittografati* e accodati uno alla volta all'ultimo blocco della lista. Ogni blocco contiene anche dati relativi a tutti i suoi dettagli, tra cui il proprietario, e ad ogni operazione di lettura e riscrittura viene controllato che il blocco in questione appartenga all'utente dell'operazione in modo da assicurare l'accesso ad ogni utente solo ai propri blocchi.
  Esistono molte *Blockchain*, e solitamente risiedono in spazi *cloud* o *server* condivisi e distribuiti in modo da essere di pubblico accesso, ovvero utilizzabili da tutti, di politiche trasparenti, cioè che sono note e dimostrate il rispetto delle proprie regole di funzionamento, affidabili, quindi sempre disponibili, e memorizzati nonostante le loro notevoli dimensioni. Si gestiscono autonomamente, pertanto non vi sono amministratori, eliminando così pericoli di eccezioni o interventi umani, rischiosi in quanto potrebbero invalidare le politiche di funzionamento e compromettere il servizio. Possono essere usate contemporaneamente da più piattaforme decentralizzate *Web3.0* compatibili. Per il progetto viene implementato la possibilità di usare tutte le *Blockchain* compatibili con le piattaforme decentralizzate *Web3.0* scelte [B1].
  La *crittografia* dei dati avviene con la chiave del proprietario, e permette di ricavare i dati originali. Questa chiave è segreta, quindi ne viene salvata una copia *crittografata* nel relativo blocco per confrontarla con l'originale ad ogni operazione di lettura e riscrittura. La copia *crittografata* della chiave è ricavata sfruttando una funzione di *Hash* pubblica in quanto non reversibile, ovvero che non permette di ricavare il valore iniziale del dato. Non sono ammesse rimozioni di dati, in quanto bisognerebbe rimuovere tutto il relativo blocco, eliminare il dato in questione e inserire nuovamente il blocco alla stessa posizione, ovviamente prima staccando tutti i blocchi successivi, e poi riattaccandoli. Nel frattempo, però, potrebbero essere inseriti nuovi blocchi, creando anomalie nella struttura della *Blockchain* in quanto l'ultimo blocco attualmente in lista non coinciderebbe con quello reale. Bisognerebbe quindi bloccare l'intero sistema a nuovi caricamenti in questi frangenti, ma questa pratica non è sensata in quanto potrebbe rallentare eccessivamente l'uso del sistema e creare una coda di caricamenti troppo grande e pesante [B4].

- **Piattaforma decentralizzata *Web3.0*:** sistema in grado di eseguire tutte le operazioni su *Blockchain* sopra descritte. Esistono molte piattaforme decentralizzate *Web3.0* e ognuna può usare contemporaneamente più *Blockchain* compatibili [B2]. Ogni piattaforma ha la propria *criptovaluta*, ovvero una moneta con cui avvengono le comunicazioni con il sistema e il cui valore dipende da quanto viene utilizzata la piattaforma stessa, in quanto ogni comunicazione ha un costo tale da comportare un ricavo alla piattaforma, aumentando il suo, e quindi indirettamente anche quello della sua moneta, valore. Ogni comunicazione richiede infatti il versamento di determinato ammontare di tale *criptovaluta* dall'utente mittente in base alla complessità computazionale, cioè risorse in termini di tempo e denaro per processare un'azione, utilizzata dall'operazione richiesta [B7]. Per il progetto viene utilizzato *Ethereum* in quanto, oltre ad essere la piattaforma decentralizzata *Web3.0* più supportata e affidabile, cioè sempre disponibile, è anche quella di riferimento per tutte le altre simili e permette di lavorare con *Smart Contracts*. Viene anche utilizzato *Ganache*, applicazione *desktop* che permette di creare localmente una piattaforma decentralizzata *Web3.0* di tipo *Ethereum* con relativa *Blockchain.*

  Mentre le piatteforme normali consentono solamente di gestire informazioni relative alla propria *criptovaluta*, salvando dati inerenti all'ammontare di denaro di ogni utente e le loro transazioni, quelle di tipo *Ethereum* permettono di lavorare con qualunque tipo di dato. A tal proposito però, occorre che la piattaforma sappia in che modo gestire ogni tipo di dato. Ad esempio, una piattaforma sa che a ogni transazione corrisponde una variazione di *criptovaluta* posseduto da uno o più utenti e un incremento del valore della *criptovaluta* stessa. Deve quindi calcolare e impostare il nuovo ammontare di denaro posseduto da questi utenti e valore della propria *criptovaluta. C*on altri tipi di dati invece, la piattaforma non sa a cosa corrispondano e quindi quali azioni eseguire [B8]. Si utilizzano quindi *Smart Contracts*, contratti definiti da un utente per indicare come alcuni dei suoi tipi di dato devono essere gestiti dalla piattaforma decentralizzata *Web3.0* utilizzata. Questi contratti sono interni alle applicazioni che usano *Blockchain*, in quanto definiscono le funzionalità implementate e indicano in che modo devono essere gestiti i dati, ma una copia deve sempre essere caricata anche nella piattaforma usata una sola volta e precedentemente all'avvio dell'applicazione. Questa migrazione avviene da riga di comando per mano di un amministratore di sistema, seguendo la stessa sequenza d'interazioni di quelle normali, ma automatizzandone il processo in *background* per non richiedere alcuna interazione intermedia dell'utente [B11].
- ***Provider*:** per comunicare con le *Blockchain* bisogna utilizzare indirizzi pubblici di rete in cui esse risiedono. Così facendo però, soprattutto in operazioni di lettura, spesso le transazioni falliscono visto che scorrere tutti i blocchi alla ricerca di uno specifico porta a errori di limite temporale dovute alle dimensioni delle *Blockchain* [B3]. A risolvere questo problema esistono i *provider*, che intermediano la comunicazione tra le due parti, assicurando il completamento delle transazioni e velocizzando l'intero processo: l'applicazione comunica con il *provider*, che la identifica tra le sue gestite e inoltra la comunicazione al nodo specifico della *Blockchain* utilizzata dal sistema, così da limitarne il numero di blocchi con cui interagire. Per il progetto viene utilizzato *Infura*, come voluto dall'azienda in quanto già in utilizzo nell'applicazione non proprietaria in utilizzo. Il seguente schema mostra il processo con cui il *provider* inoltra le comunicazioni ad un nodo preciso della *Blockchain* [B6]:

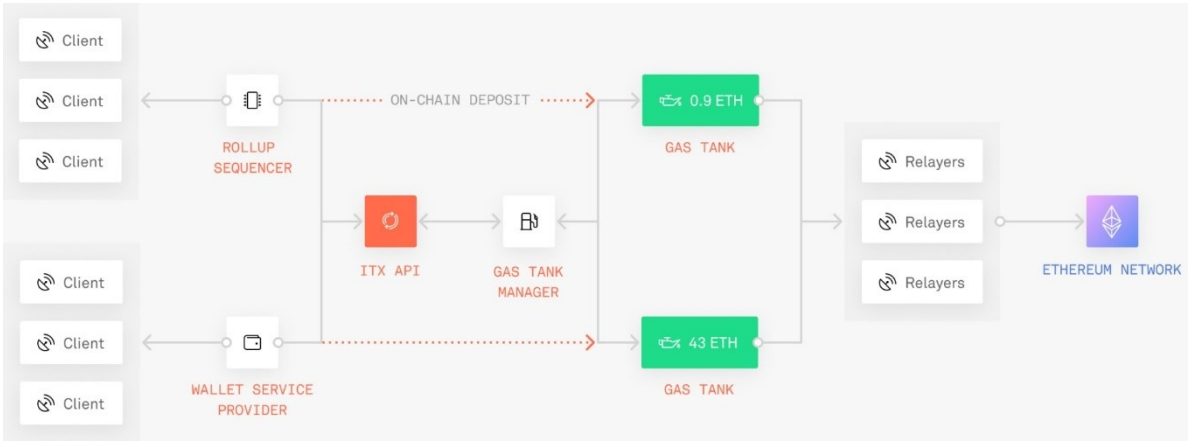

Figura 3.1: Diagramma delle interazioni tra Infura ed Ethereum

- ***Crypto wallet*:** sistema per confermare le transazioni e gestire il proprio w*allet*, ovvero portafoglio digitale, configurando il profilo, accreditando un ammontare di *criptovaluta* e configurando le reti di *Blockchain* [B3]. Per il progetto viene usato *MetaMask*, cioè l'estensione per *browser* più supportata e affidabile, quindi sempre disponibile, compatibile con le *Blockchain*, piattaforma decentralizzata *Web3.0*, *provider* e *dApp* utilizzate [B5] [B9].
- ***DApp*:** applicazione decentralizzata, ovvero in grado di operare autonomamente, con possibilità di essere usufruito da tutti e che usufruiscono di *Blockchain* [B3]. Si presentano come siti *web* distribuiti su *server*, caricati *online* o su una rete privata e che sfruttano piattaforme decentralizzate *Web3.0* per mettere a disposizione funzionalità all'utente. *AlphaNotary* ne è un esempio [B10].

Senza l'infrastruttura esterna sarebbe molto più complesso sviluppare un sistema come *AlphaNotary*, in quanto sarebbe necessario gestire internamente al sistema tutti i processi di comunicazione precedentemente descritti. Ciò metterebbe in dubbio la fattibilità realizzativa del progetto, in quanto bisognerebbe sviluppare tutte le componenti dell'apparato esterno, aumentando notevolmente così il carico di lavoro e la difficoltà realizzativa.

3.1.2 Diagramma degli oggetti delle interazioni esterne

Il seguente diagramma degli oggetti, poi descritto e spiegato approfonditamente, riassume l'insieme delle interazioni esterne:

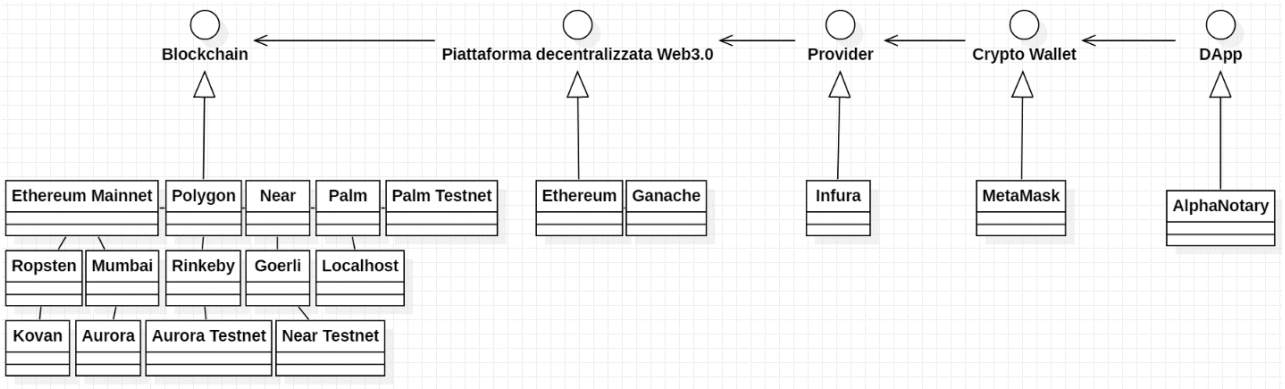

Figura 3.2: Diagramma degli oggetti delle interazioni esterne

- **Attori interni:**
  - ***DApp*: *AlphaNotary***.
  - ***Crypto wallet*: *MetaMask***.
  - ***Provider*: *Infura***.
  - **Piattaforma decentralizzata *Web3.0*:**
    - *Ethereum*.
    - *Ganache*.
  - ***Blockchain*:** quelli *Ethereum* compatibili sono:
    - ***Ethereum Mainnet*:** *Blockchain* principale di *Ethereum*.
    - ***Ropsten*:** *Blockchain* di prova principale di *Ethreum*.
    - ***Polygon*:** *Blockchain* secondaria di *Ethreum*.
    - ***Palm*:** *Blockchain* secondaria di *Ethreum*.
    - ***Aurora*:** *Blockchain* secondaria di *Ethreum*.
    - ***Near*:** *Blockchain* secondaria di *Ethreum*.
    - ***Mumbai*:** *Blockchain* di prova principale di *Polygon*.
    - ***Localhost*:** *Blockchain* di prova locale di *Ganache*.
    - ***Kovan*:** *Blockchain* di prova secondaria di *Ethreum*.
    - ***Rinkeby*:** *Blockchain* di prova secondaria di *Ethreum*.
    - ***Goerli*:** *Blockchain* di prova secondaria di *Ethreum*.
    - ***Palm Testnet*:** *Blockchain* di prova primaria di *Palm*.
    - ***Aurora Testnet*:** *Blockchain* di prova primaria di *Aurora*.
    - ***Near Testnet*:** *Blockchain* di prova principale di *Near*.
- **Attore esterno:** un generico utente del prodotto.
- **Precondizioni:** l'utente ha impostato correttamente:
  1. Il sistema esterno.
  2. Il progetto e avviato l'applicazione.
  3. Vuole *notarizzare* un *NFT*.
- **Postcondizione:** l'utente ha *notarizzato* un *NFT*.
- **Scenario principale:**
  1. *AlphaNotary* invia al momento della compilazione gli *Smart Contracts* a *MetaMask*, che crea una transazione verso *Infura*.
  2. *Infura* lo inoltra alla piattaforma decentralizzata *Web3.0* attualmente impostata.
  3. Quando l'utente vuole eseguire un'operazione di *notarizzazione*, *AlphaNotary* invia un'azione alla piattaforma decentralizzata *Web3.0* attualmente impostata seguendo lo stesso percorso.
  4. La piattaforma decentralizzata *Web3.0* attualmente impostata utilizza gli *Smart Contracts* relativi al programma per eseguire l'azione indicata, salva il risultato sulla *Blockchain* indicata e ne restituisce l'esito ripercorrendo a ritroso la sequenza.

3.1.3 Diagramma di sequenza delle interazioni esterne

Il seguente diagramma di sequenza invece, ricavato dal diagramma degli oggetti e anch'esso poi descritto e spiegato approfonditamente, mostra la sequenza delle interazioni del sistema esterno:

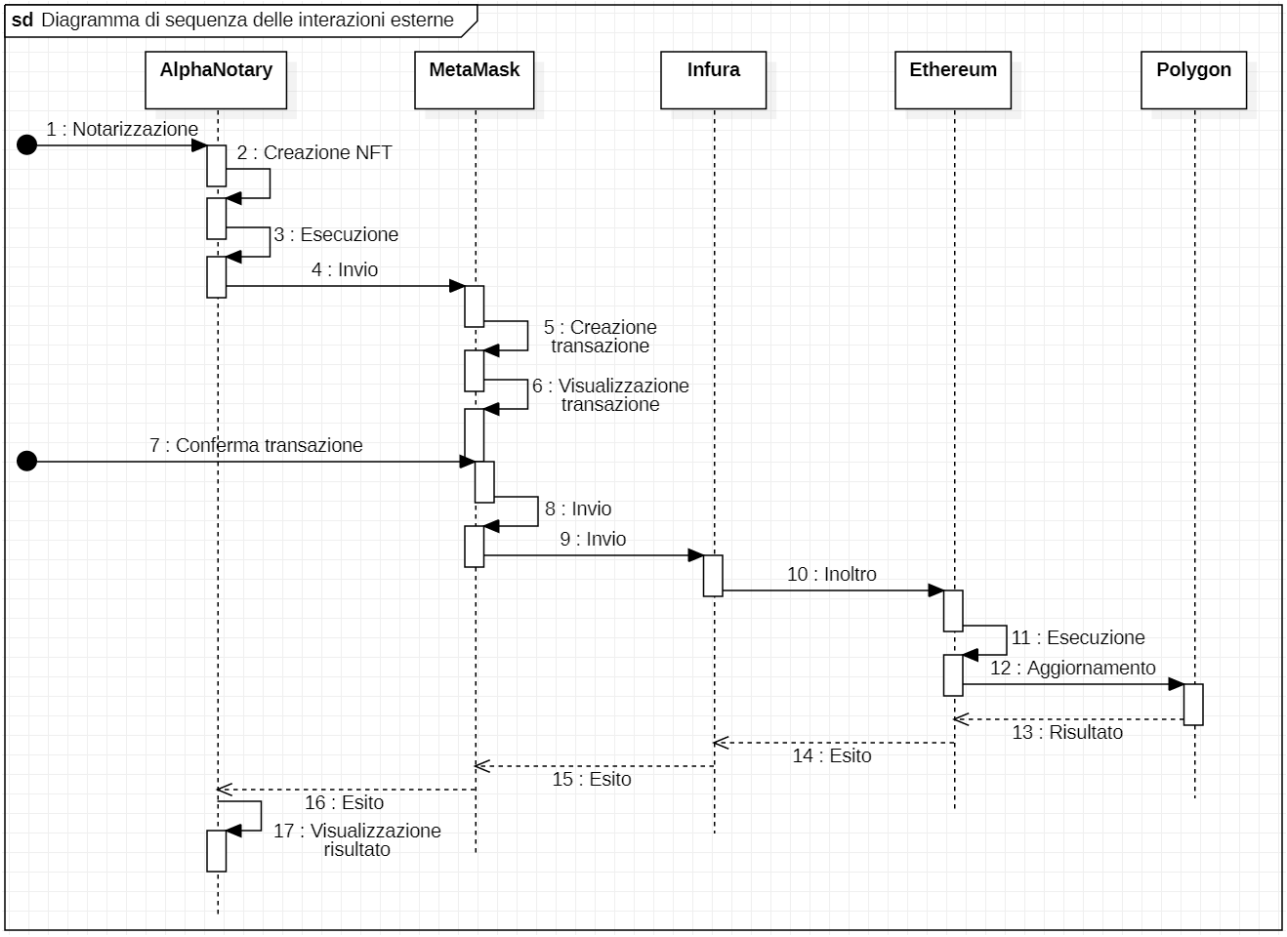

Figura 3.3: Diagramma di sequenza delle interazioni esterne

- **Attori interni:**
  - *AlphaNotary*.
  - *MetaMask*.
  - *Infura*.
  - *Ethereum*.
  - *Polygon*.
- **Attore esterno:** un generico utente del prodotto.
- **Precondizioni:** l'utente ha impostato correttamente:
  1. Il sistema esterno.
  2. Il progetto e avviato l'applicazione.
  3. Vuole *notarizzare* un *NFT*.
- **Postcondizione:** l'utente ha *notarizzato* un *NFT*.
- **Scenario principale:**
  1. L'utente chiede all'applicazione di eseguire un'operazione di *notarizzazione*.
  2. Il sistema crea un *NFT* a partire dai dati forniti ed esegue l'operazione, il cui primo passo consiste nell'inviare le informazioni ricavate a *MetaMask*.

3. *MetaMask* crea quindi una transazione con i dati ricevuti e la visualizza nell'interfaccia utente in attesa di conferma.
4. Una volta confermata, la transazione viene spedita a *Infura*, che la inoltra a *Ethereum*.
5. *Ethereum* esegue l'operazione e salva i risultati su *Polygon*.
6. Il risultato dell'operazione viene restituito all'applicazione ripercorrendo a ritroso la sequenza delle interazioni a partire da *Polygon*.
7. L'applicazione elabora e visualizza una risposta graficamente adatta all'utente.

- **Estensione:** Il fallimento dell'operazione di *notarizzazione* visualizza un messaggio di errore durante la sua esecuzione, specificandone i dettagli e le cause.

3.2 Interfaccia d'uso

3.2.1 Diagramma dei casi d'uso

Il seguente diagramma dei casi d'uso, poi descritto e spiegato approfonditamente, mostra un utilizzo completo del sistema, che quindi coinvolge tutte le azioni eseguibili e soddisfa tutti i requisiti imposti per il progetto:

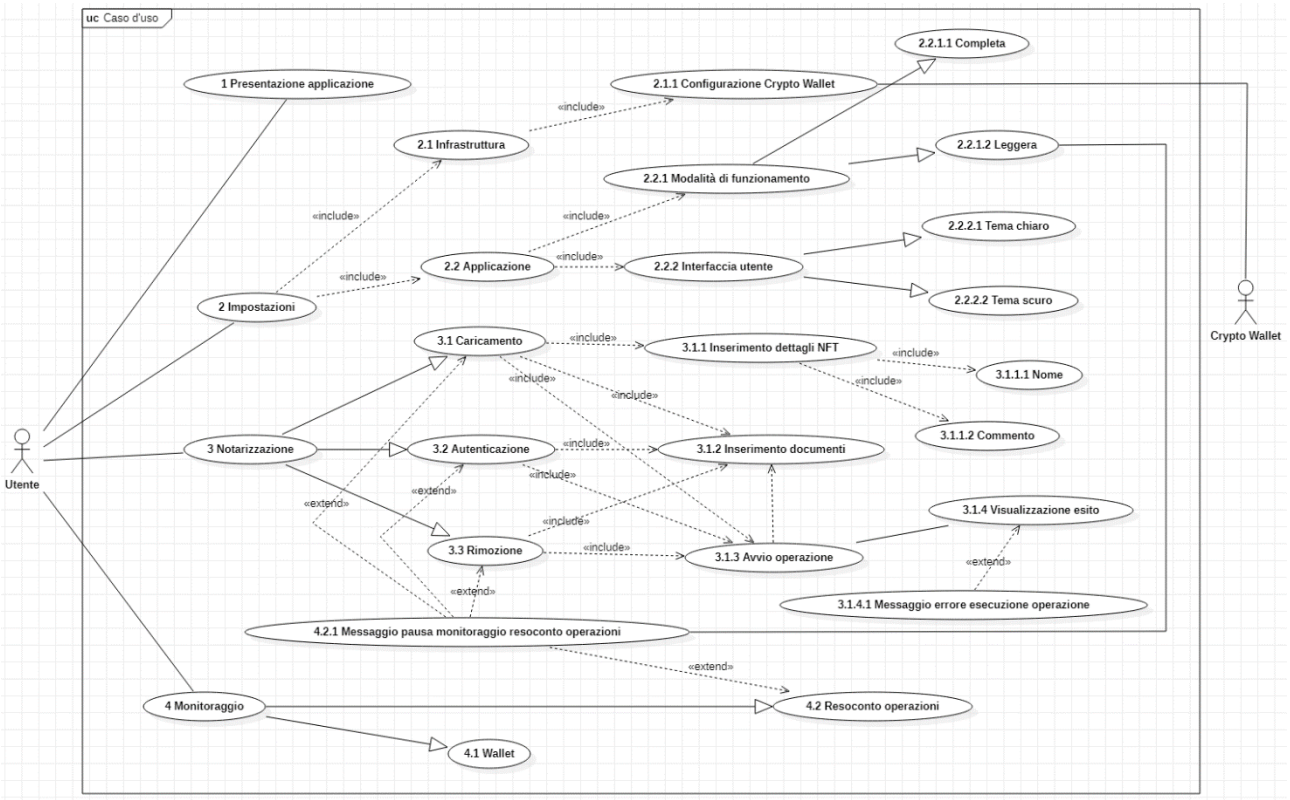

Figura 3.4: Diagramma dei casi d'uso

- **Attori interni:**
  - Utente, un generico cliente del prodotto.
  - *AlphaNotary*.
- **Attore esterno:** *il cripto wallet*.
- **Precondizioni:** l'utente ha impostato correttamente:
  1. Il sistema esterno.
  2. Il progetto e avviato l'applicazione.
  3. Vuole *notarizzare* e monitorare *NFT*.

- **Postcondizione:** l'utente ha *notarizzato* e monitorato *NFT*.
- **Scenario principale:**
    1. L'utente visualizza la presentazione applicazione, utile ad apprendere i dettagli del sistema e come usarlo.
    2. L'utente può procedere all'impostazione del sistema configurando:
        - **Infrastruttura:** per completare, obbligatorio per usare l'applicazione, o modificare la configurazione del sistema esterno usando il *cripto wallet*.
        - **Applicazione:** per modificare il comportamento o l'aspetto del sistema, con possibilità di scegliere tra:
            - **Modalità di funzionamento:** decidendo tra:
                - **Leggera:** mette in pausa il monitoraggio a vantaggio delle prestazioni del sistema.
                - **Completa:** attiva il monitoraggio a scapito delle prestazioni del sistema. È l'impostazione predefinita.
            - **Interfaccia utente:** decidendo tra:
                - **Tema chiaro:** imposta l'insieme dei colori dell'applicazione ad una gamma più chiara che aumenta la luminosità dell'interfaccia utente. È l'impostazione predefinita.
                - **Tema scuro:** imposta l'insieme dei colori dell'applicazione ad una gamma più scura che diminuisce la luminosità dell'interfaccia utente.
    4. L'utente può procedere alla *notarizzazione* eseguendo operazioni di:
        1. **Caricamento:** che comprende le seguenti mosse:
            1. Inserimento dettagli *NFT*, ovvero nome e commento (facoltativo).
            2. Inserimento uno o più documenti.
            3. Avvio operazione.
            4. Visualizzazione esito.
        2. **Autenticazione:** uguale a all'operazione di caricamento ma senza inserire i dettagli dell'*NFT* da creare.
        3. **Rimozione:** uguale a all'operazione di caricamento ma senza inserire i dettagli dell'*NFT* da creare.
    5. L'utente può procedere al monitoraggio attraverso:
        - ***Wallet*:** riassume il *wallet* mostrando:
            - Codice identificativo del *wallet*.
            - Nome della piattaforma decentralizzata *Web3.0* che attualmente sta usando il *wallet*.
            - Applicazione che attualmente sta usando il *wallet*.
            - Saldo rimanente nel *wallet*.
            - *Il cripto wallet* che attualmente sta usando il *wallet*.
            - *Provider* che attualmente sta usando il *wallet*.
        - **Resoconto delle operazioni:** mostra le operazioni di *notarizzazione* in ordine cronologico, riportando per ognuno:
            - Tipo di operazione.
            - Nome dell' *NFT*.

        - Eventuale commento dell'*NFT*.
        - Codice del documento dell'*NFT*.
        - Proprietario dell'*NFT*.
        - Data di creazione dell'*NFT*.
        - Proprietario dell'operazione.
        - Data d'esecuzione.
- **Estensioni:**
    - L'attivazione della modalità leggera visualizza un messaggio in tutti i campi di *notarizzazione* e nel resoconto operazioni che avverte l'utente che il monitoraggio è in pausa.
    - Il fallimento dell'operazione di *notarizzazione* visualizza un messaggio di errore durante la sua esecuzione, specificandone i dettagli e le cause.
    - La configurazione sbagliata del *cripto wallet* visualizza un messaggio di errore di connessione al sistema esterno.

3.2.2 Diagramma delle attività

Il seguente diagramma delle attività invece, ricavato dal diagramma dei casi d'uso e anch'esso poi descritto e spiegato approfonditamente, mostra l'ordine delle azioni:

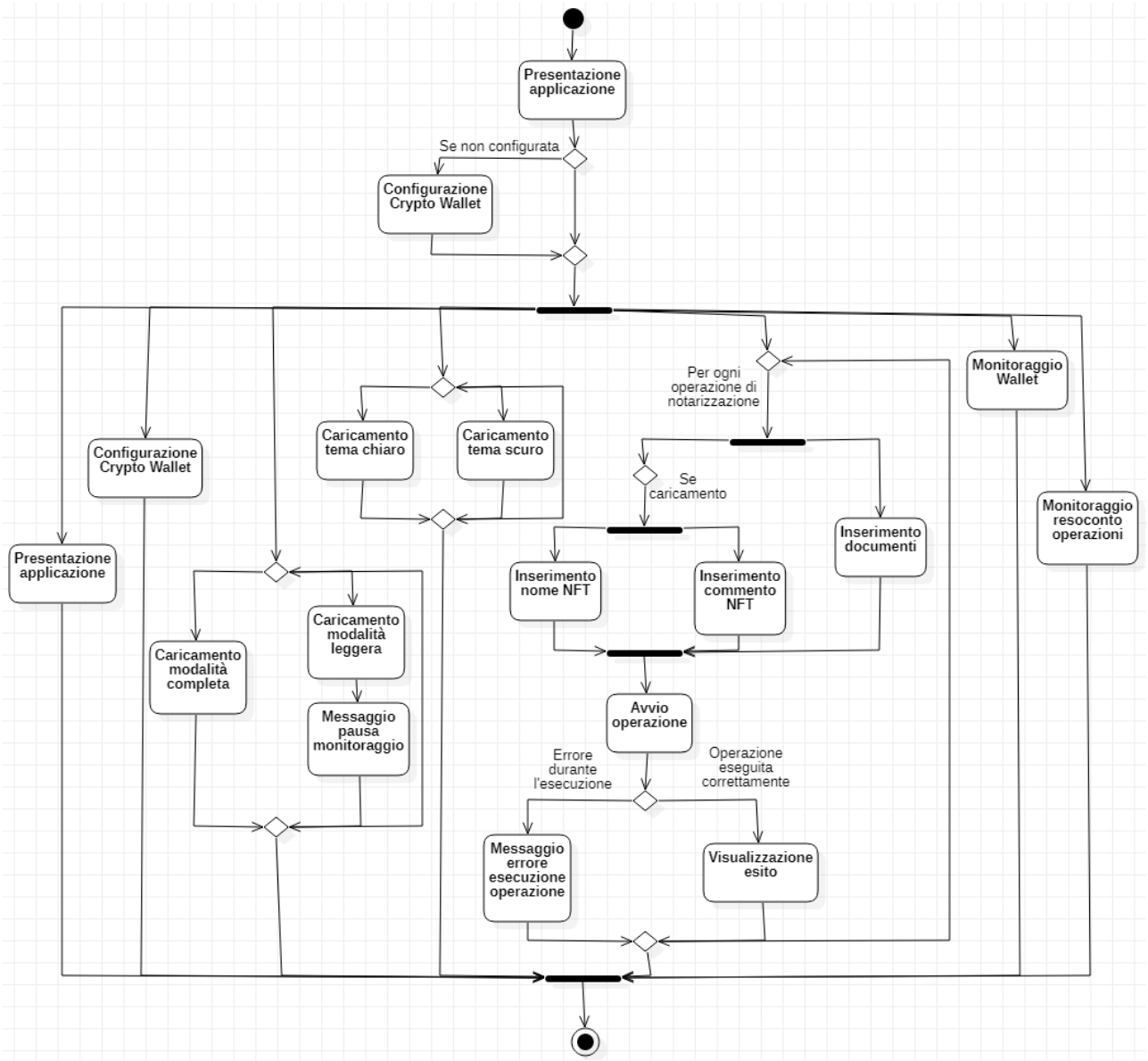

Figura 3.5: Diagramma delle attività

- **Attori interni:**
  1. Utente, un generico cliente del prodotto.
  2. *AlphaNotary*.
- **Attore esterno:** il *cripto wallet*.
- **Precondizioni:** l'tente ha impostato correttamente:
  1. Il sistema esterno.
  2. Il progetto e avviato l'applicazione.
  3. Vuole *notarizzare* e monitorare *NFT*.
- **Postcondizioni:** l'utente ha *notarizzato* e monitorato *NFT*.
- **Scenario principale:**

1. L'utente visualizza la presentazione applicazione, utile ad apprendere i dettagli del sistema e come usarlo.
2. Se non è ancora stata configurata, l'utente deve procedere alla configurazione della *crypro wallet* tramite l'apposito sistema.
3. Solo adesso l'applicazione mette a disposizione le seguenti funzionalità, che possono anche essere eseguite contemporaneamente, in quanto sono indipendenti tra loro, e molteplici volte:
    - Visualizzare nuovamente la presentazione applicazione.
    - Modificare la configurazione del *cripto wallet* tramite l'apposito sistema.
    - Impostare la modalità di funzionamento del sistema scegliendo tra il caricamento della modalità completa o leggera, in questo caso viene visualizzato un messaggio in tutti i campi di *notarizzazione* e nel resoconto operazioni che avverte l'utente che il monitoraggio è in pausa.
    - Impostare l'interfaccia utente scegliendo tra il caricamento del tema chiaro o del tema scuro.
    - *Notarizzare* eseguendo operazioni di:
        1. **Caricamento:** che comprende le seguenti mosse:
            1. Inserimento dettagli *NFT*, ovvero nome e commento (facoltativo).
            2. Inserimento uno o più documenti.
            3. Avvio operazione.
            4. Visualizzazione esito.
        2. **Autenticazione:** uguale a all'operazione di caricamento ma senza inserire i dettagli dell'*NFT* da creare.
        3. **Rimozione:** uguale a all'operazione di caricamento ma senza inserire i dettagli dell'*NFT* da creare.
    - Monitorare il *wallet*, riassunto mostrando:
        - Codice identificativo del *wallet*.
        - Nome della piattaforma decentralizzata *Web3.0* che attualmente sta usando il *wallet*.
        - Applicazione che attualmente sta usando il *wallet*.
        - Saldo rimanente nel *wallet*.
        - *Il cripto wallet* che attualmente sta usando il *wallet*.
        - *Provider* che attualmente sta usando il *wallet*.
    - Monitorare il resoconto delle operazioni, che mostra le operazioni di *notarizzazione* in ordine cronologico, riportando per ognuno:
        - Tipo di operazione.
        - Nome dell' *NFT*.
        - Eventuale commento dell'*NFT*.
        - Codice del documento dell'*NFT*.
        - Proprietario dell'*NFT*.
        - Data di creazione dell'*NFT*.
        - Proprietario dell'operazione.
        - Data d'esecuzione.

3.3 Analisi dei requisiti

Il seguente elenco rappresenta la lista completa dei requisiti specifici del progetto, ottenuti a partire dai requisiti richiesti dall'azienda in cui quelli relativi al sistema sono stati approfonditi in base di quanto progettato nel diagramma precedenti:

- **Obbligatorie:** requisiti primari, necessari alla buona riuscita dello stage.
    1. Visualizzazione del resoconto delle operazioni.
    2. Configurazione del *cripto wallet*.
    3. Operazione di caricamento.
    4. Operazione di autenticazione.
    5. Operazione di rimozione.
    6. Inserimento nome *NFT*.
    7. Inserimento commento *NFT*.
    8. Inserimento documenti *NFT*.
    9. Avvio operazione di *notarizzazione*.
    10. Visualizzazione esito operazione di *notarizzazione*.
    11. Utilizzo delle seguenti tecnologie:
        - *Polygon*.
        - *Ethereum*.
        - *Infura*.
        - *MetaMask*.
- **Desiderabili:** non necessari, ma che contribuiscono alla completezza del prodotto.
    1. Messaggio di errore esecuzione operazione a seguito del fallimento di un'operazione di *notarizzazione*.
    2. Modalità leggera/modalità completa.
    3. Messaggio di pausa monitoraggio resoconto operazioni a seguito dell'attivazione della modalità leggera.
    4. Visualizzazione del proprio *wallet*.
    5. Documentazione della *dApp* di *notarizzazione*.
    6. Documentazione dell'interfaccia di monitoraggio.
    7. Utilizzo delle seguenti tecnologie:
        - *Blockchain Ethereum* compatibili.
        - *Ganache*.
- **Opzionali:** puramente superflui, che portano valore aggiunto al progetto.
    1. Pagina di presentazione dell'applicazione.
    2. Tema scuro/tema chiaro.
    3. Documentazione delle funzionalità complementari.

3.4 Infrastruttura interna

3.4.1 Architettura del sistema

La realizzazione del prodotto è stata preceduta da una lunga fase di progettazione, in cui sono state definite le caratteristiche strutturali che il sistema avrebbe dovuto avere, al fine di garantirne la qualità, uno sviluppo semplice e la possibilità di evolvibilità futura. Innanzitutto, il sistema presenta un'architettura, ovvero un insieme di relazioni tra le macro-componenti dell'infrastruttura, monolitica, cioè in cui tutto il programma risiede dentro un unico pacchetto. Questa scelta, che deriva dal fatto che l'applicazione non prevede componenti distribuite, semplifica di molto la realizzazione e la distribuzione del sistema [S10].

Al suo interno, viene delineato un'architettura a *layer*, ovvero in cui la struttura del sistema è fatta a livelli, ciascuna formata da componenti appartenenti ad uno stesso processo. Le componenti di comunicano solo con quelli dello stesso o quello strettamente sottostante. In questo modo si organizza la struttura, si diminuisce l'accoppiamento e si minimizza la complessità di gestione delle componenti del sistema.

In quest'architettura, tutti i livelli dipendono indirettamente da quello più basso, solitamente rappresentato dal *database*, ma in questo caso costituito dalle *Blockchain*, in quanto svolgono la stessa funzione. Il seguente diagramma riassume l'architettura a *layer [S11]*:

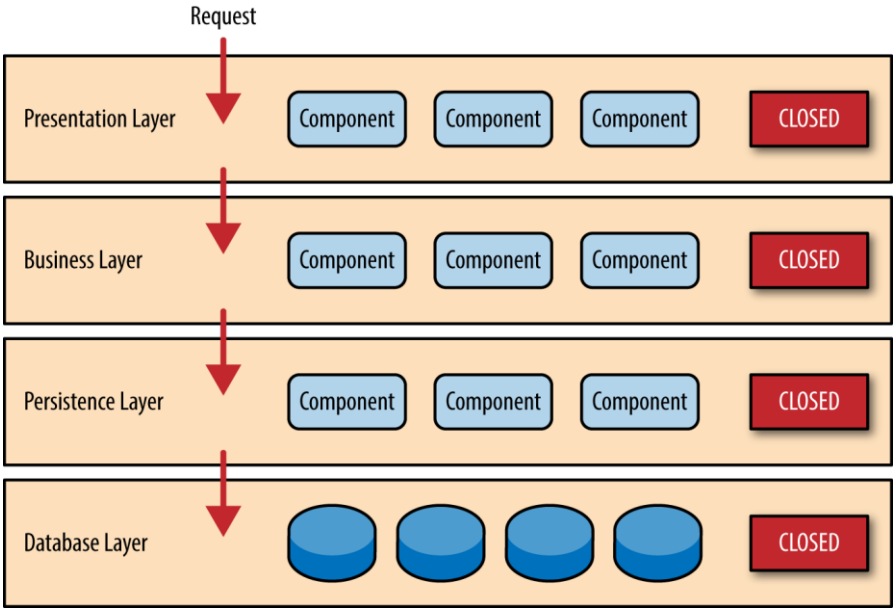

Figura 3.6: Diagramma dell'architettura a layer

L'architettura prevede l'utilizzo di alcuni *design pattern*, ovvero descrizioni di modelli logici che costituiscono soluzioni progettuali generali per risolvere problemi ricorrenti nella progettazione, sviluppo o implementazione di sistemi e ridurre così anche il debito tecnico del lavoro.

Si applicano se viene usata la programmazione orientata agli oggetti in quanto mostrano relazioni tra classi o oggetti, definendo la struttura totale o parziale del sistema. Sono stati utilizzati i seguenti *design pattern*, raggruppati per categoria, in modo da risolvere i problemi di realizzazione dell'applicazione previsti, assicurando maggiori benefici rispetto alle conseguenze introdotte [S12]:

- **Architetturali:** separano il comportamento delle componenti dalla risoluzione delle loro dipendenze. Collegano in modo esplicito più componenti insieme aumentandone l'accoppiamento e diventando quindi più difficili da testare e mantenere. Sono stati usati i seguenti *design pattern* architetturali:
    - ***Dependency Injection*:** fornisce tecniche per minimizzare il grado di dipendenza tra le componenti di un sistema mediante il passaggio delle dipendenze tra classi. Il seguente diagramma delle classi mostra il *design pattern*:

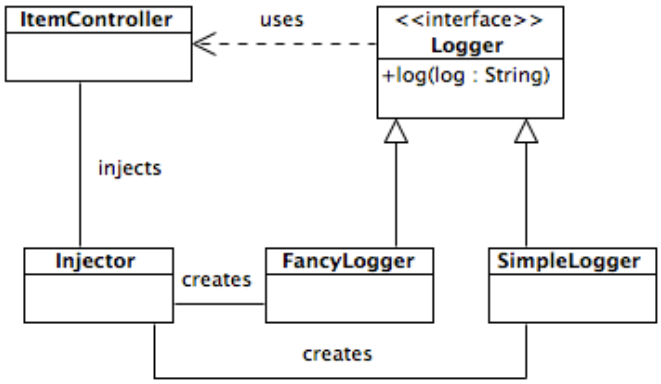

Figura3.7: Diagramma delle classi desgin pattern Dependency Injection

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

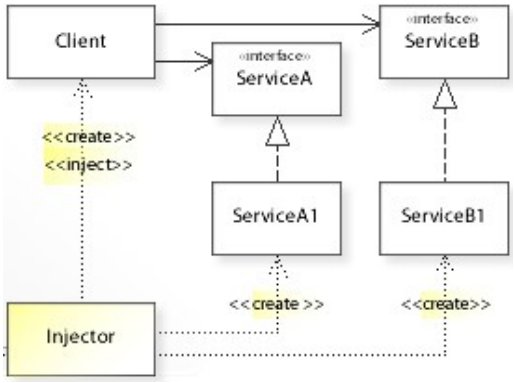

Figura3.8: Diagramma di sequenza design pattern Dependency Injection

    - ***MVP* (*Model View Presenter*):** appartiene alla famiglia dei *design pattern MV** (*MVStar*), tutte con le stesse componenti, ma con diverse modalità di comunicazione. Viene utilizzato in applicazioni con un'interfaccia utente per permette di organizzare e ridurre le dipendenze tra le seguenti componenti mediante la definizione di ruoli per ognuna di esse:
        - ***Model*:** risponde al *presenter* con i risultati delle funzionalità del programma da esso richieste. È indipendente dal resto del sistema, in quanto che non necessita di altre componenti per svolgere il suo incarico.
        - ***View*:** rappresenta l'interfaccia utente con cui l'utente interagisce. All'avvenire di determinate azioni nella grafica, comunica con il *presenter* e poi visualizza i risultati da esso fornitigli.
    - ***Presenter*:** gestisce il servizio commutando le interazioni dell'utente nell'interfaccia grafica, fornite dalla *view*, in azioni da eseguire nel *model*, di cui elabora il significato per indicare alla *view* i dati da visualizzare.

Il seguente diagramma delle classi mostra il *design pattern*:

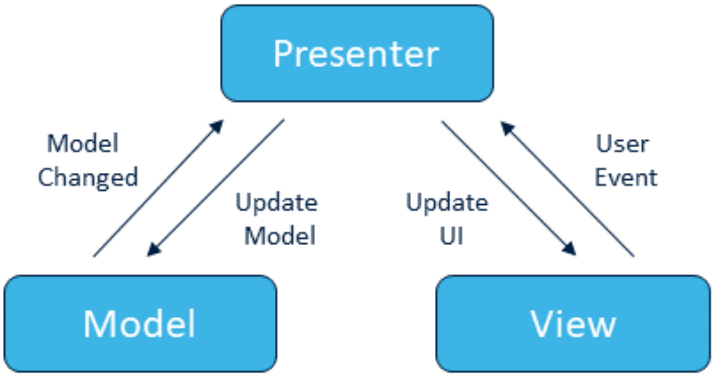

Figura 3.4: Diagramma delle classi design pattern MVP

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

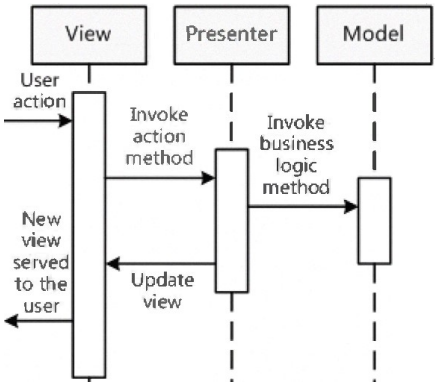

Figura 3.10: Diagramma di sequenza design pattern MVP

- **Creazionali:** risolvono problemi relativi alla creazione di oggetti. Si fa un ampio uso delle interfacce in modo da rendere il sistema indipendente dall'implementazione delle sue componenti. Sono stati usati i seguenti *design pattern* creazionali:
  - **Abstract Factory:** fornisce un'interfaccia per creare famiglie di oggetti senza dover specificare classi concrete. Vengono definite le interfacce degli oggetti da creare. Viene usato per fornire librerie di classi senza rivelarne l'implementazione oppure per configurare sistemi di più famiglie di oggetti. Il seguente diagramma delle classi mostra il *design pattern*:

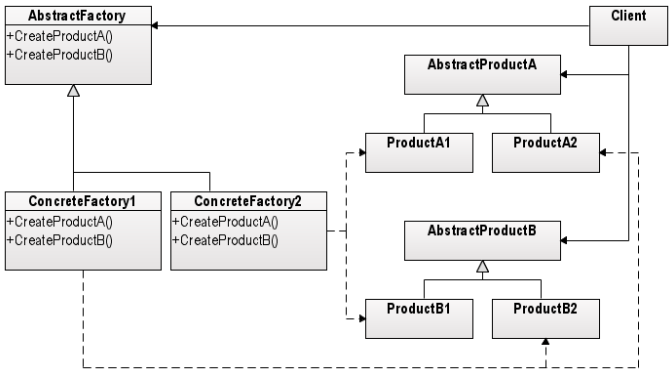

Figura 3.11: Diagramma delle classi design pattern Abstract Factory

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

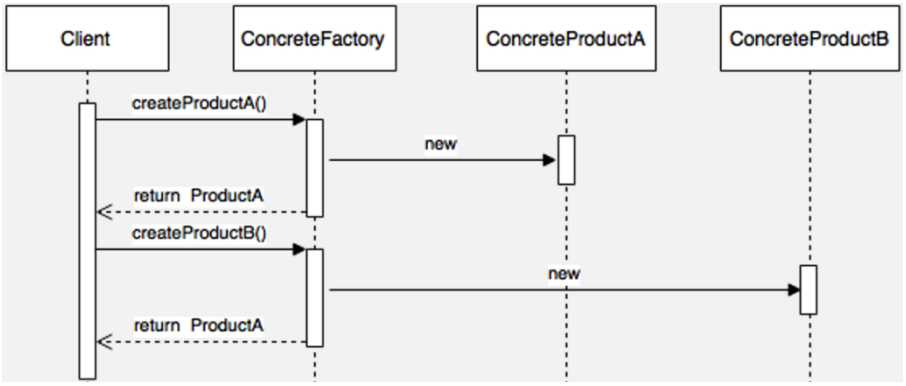

Figura 3.12: Diagramma di sequenza design pattern Abstract Factory

- **Comportamentali:** definiscono il modo in cui le classi svolgono le proprie funzioni e comunicano fra di loro. Sono stati usati i seguenti *design pattern* comportamentali:
  - ***Observer*:** segnalano le modifiche su un oggetto a tutti quelli ad esso relazionati, che, aggiornando di conseguenza il sistema, possono garantirne la consistenza. È composta dai seguenti due componenti:
    - ***Subject*:** classe in cui possono essere effettuate modifiche.
    - ***Observer*:** classe che osserva il *subject* e avverte tutte quelle ad esso relazionate di eventuali modifiche.

  Il seguente diagramma delle classi mostra il *design pattern*:

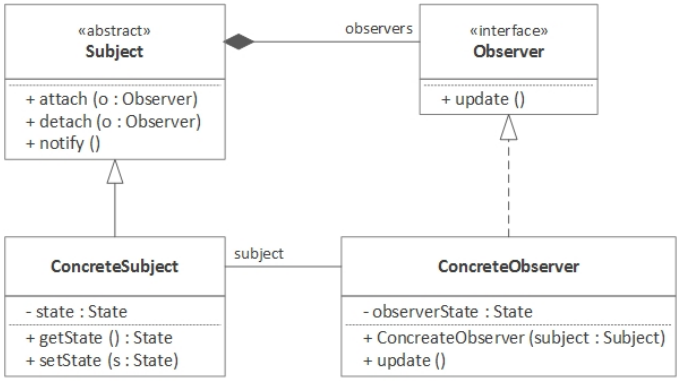

Figura 3.13: Diagramma delle classi design pattern Observer

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

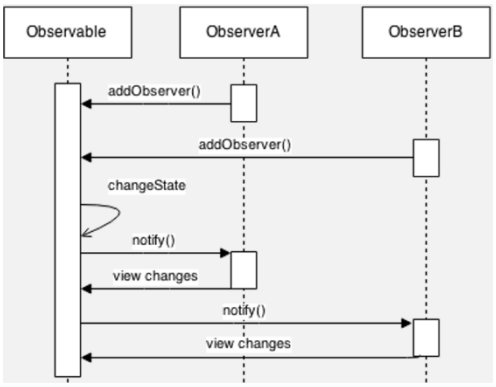

Figura 3.14: Diagramma di sequenza design pattern Observer

- ***Template Method*:** definisce lo scheletro di un algoritmo ma lascia l'implementazione di alcune sue parti alle classi derivanti, in modo da ottenere algoritmi relativi ad uno stesso processo ma che eseguono operazioni differenti. Il seguente diagramma delle classi mostra il *design pattern*:

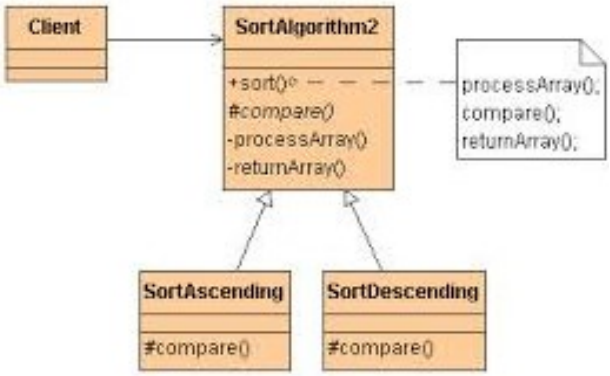

Figura 3.15: Diagramma delle classi design pattern Template Method

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

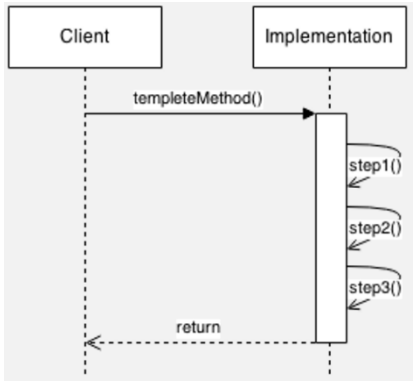

Figura 3.16: Diagramma di sequenza design pattern Template Method

- ***Strategy*:** definisce una famiglia di algoritmi tra di loro interscambiabili, in modo da avere classi che differiscono per il comportamento e non per la struttura. Si può così cambiare algoritmo durante l'esecuzione del programma tramite il polimorfismo. Il seguente diagramma delle classi mostra il *design pattern*:

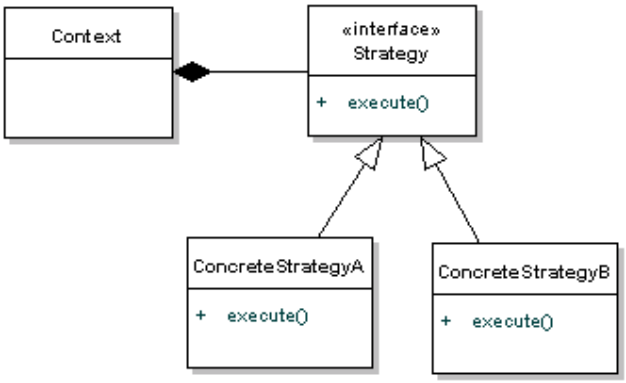

Figura 3.17: Diagramma delle classi design pattern Strategy

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

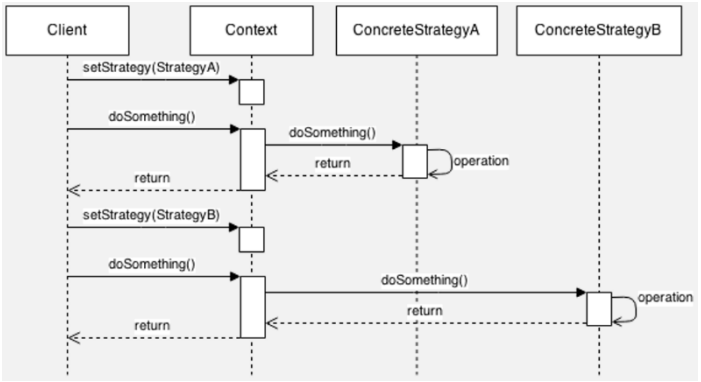

Figura 3.18: Diagramma di sequenza design pattern Strategy

- **Concorrenziali:** mantengono sincronizzato lo stato di dati condivisi in situazioni di processi che eseguono contemporaneamente attività su di esse. Sono stati usati i seguenti *design pattern* concorrenziali:
    - ***Event Based Asynchronous*:** permette di eseguire in modo asincrono operazioni scatenati da determinati eventi, in modo da aumentare l'efficienza del sistema. Il seguente diagramma delle classi mostra il *design pattern*:

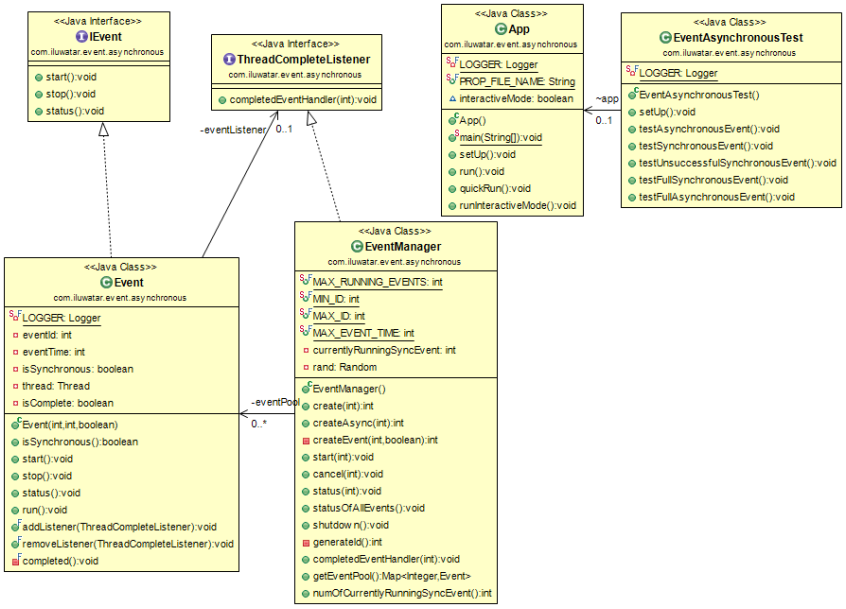

Figura 3.19: Diagramma delle classi design pattern Event Based Asynchronous

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

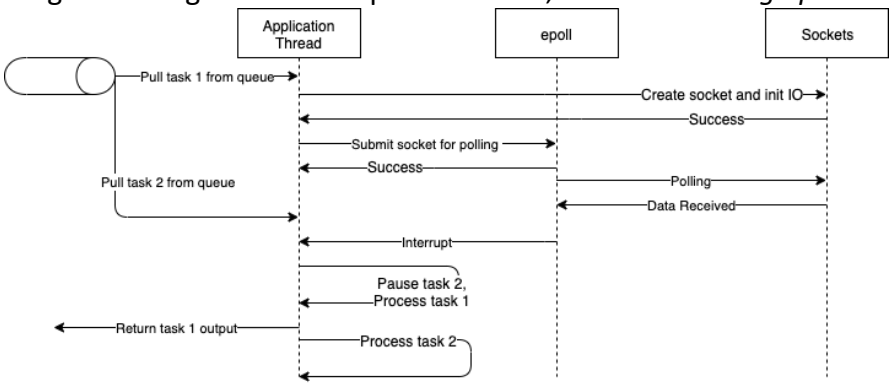

Figura 3.20: Diagramma di sequenza design pattern Event Base Asynchronous

- **Strutturali:** risolvono problemi relativi alla composizione delle classi, sfruttando l'ereditarietà e l'aggregazione in modo da consentire il riutilizzo di quelle già esistenti. Sono stati usati i seguenti *design pattern* strutturali:
    - ***Proxy*:** fornisce un oggetto controllandone l'accesso al fine di rinviarne la creazione solamente al momento dell'utilizzo, distribuendo così il lavoro ed evitando azioni inutili nel caso in cui l'oggetto non venga mai usato. Il seguente diagramma delle classi mostra il *design pattern*:

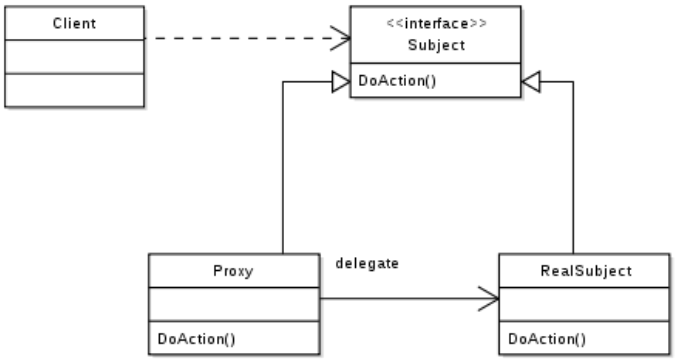

Figura 6.21: Diagramma delle classi design pattern Proxy

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

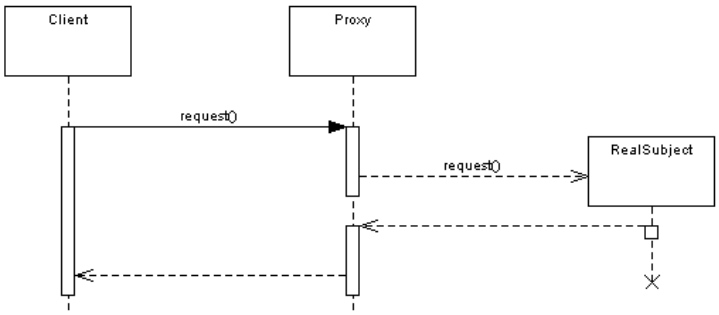

Figura 3.27: Diagramma di sequenza design pattern Proxy

    - ***Facade*:** intermedia la comunicazione con un insieme di classi fornendo un'unica interfaccia semplice da utilizzare. Rappresenta un *Single Failure Point*, ovvero unico punto di vulnerabilità, in quanto è la posizione di contatto fra diverse classi. Il seguente diagramma delle classi mostra il *design pattern*:

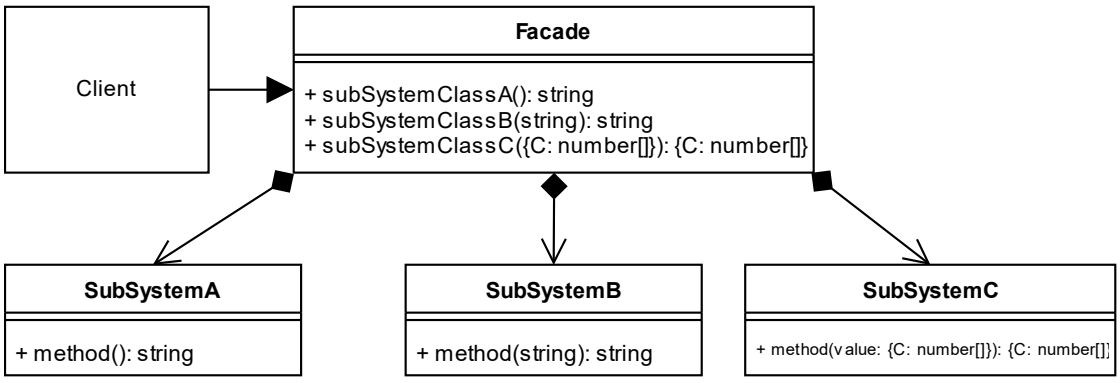

Figura3.23: Diagramma delle classi design pattern Facade

Il seguente diagramma di sequenza invece, riassume il *design pattern*:

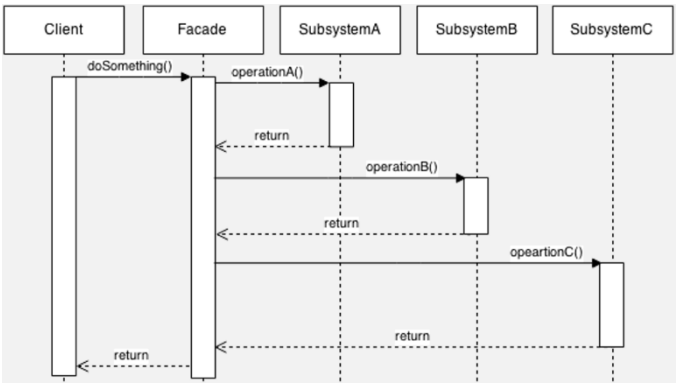

Figura 3.24: Diagramma di sequenza design pattern Facade

La progettazione dell'infrastruttura interna è inoltre stata effettuata affinché la sua realizzazione rispettasse i principi di programmazione *SOLID*, termine formato dalle iniziali dei cinque principi che la compongono e che aiutano a gestire le dipendenze in un sistema. L'intento finale di questi principi è rendere l'accoppiamento tra le componenti quanto più lasco possibile. Tra questi principi, che vengono di seguito descritti dettagliatamente, i primi tre sono fondamentali e il loro rispetto garantisce molto spesso anche la conformità agli altri [S13]:

- ***Single Responsability Principle:*** una classe deve avere una singola responsabilità, ovvero deve svolgere una sola attività. Identificare la conformità a questo principio richiede di analizzare tutto il contesto che coinvolge ogni singola classe. Il rischio di dover adattare una classe a quelle che la usano aumenta assieme al numero di responsabilità della classe stessa. L'obiettivo è avere classi piccoli e concisi. Questo principio facilita anche testare le classi.
- ***Open Closed Principle*****:** bisogna essere in grado di estendere il comportamento delle classi senza modificarle. È quindi necessario essere aperti al cambiamento ma chiusi alle modifiche, ovvero poter creare nuove funzionalità aggiungendo semplicemente codice e senza cambiare quello già esistente. La chiave è l'astrazione del codice, ottenuto dall'uso d'interfacce o classi astratte, che permettono di definire la struttura del componente ma non la sua implementazione, non implicando modifiche nel codice già esistente all'aggiunta di nuove funzionalità.
- ***Liskov Substitution Principle*****:** le classi derivate devono essere sostituibili alle proprie classi base. Nelle classi derivate, le precondizioni devono essere più forti che nelle proprie classi base. Solitamente ciò è ottenuto aggiungendo vincoli d'esecuzione ad ogni classe padre a partire dalle classi base iniziali.
- ***Interface Segregation Principle*****:** bisogna fare in modo da non essere forzati a implementare interfacce con metodi che non c'è bisogno di usare. È quindi necessario creare interfacce con solo metodi molto specifici e indispensabili a tutte le classi derivanti. Attuare il contrario rischia di portare a classi che non dispongono di tutte funzionalità da implementare, e quindi all'impossibilità di usufruire di tali interfacce.
- ***Dependency Inversion Principle*****:** bisogna fare in modo da dipendere dalle astrazioni, che definiscono le forme, e non dalle concretizzazioni, che precisano invece i comportamenti. Conviene quindi usare spesso interfacce e classi astratte, per poi implementare o ereditare da queste, anziché dipendere da una classe concreta.

Sono state usate la programmazione procedurale, funzionale e ad oggetti per garantire possibilità di riuso del codice, modularità, estensibilità e praticità. Inoltre, è stato posto particolare attenzione agli algoritmi e alle strutture dati usate al fine di garantire la massima efficienza del servizio finale. Infine, il lavoro è stato ben documentato per permetterne l'evoluzione futura dell'infrastruttura.

3.4.2 Diagramma delle classi

Il seguente diagramma delle classi, poi descritto e spiegato approfonditamente, mostra l'architettura del sistema:

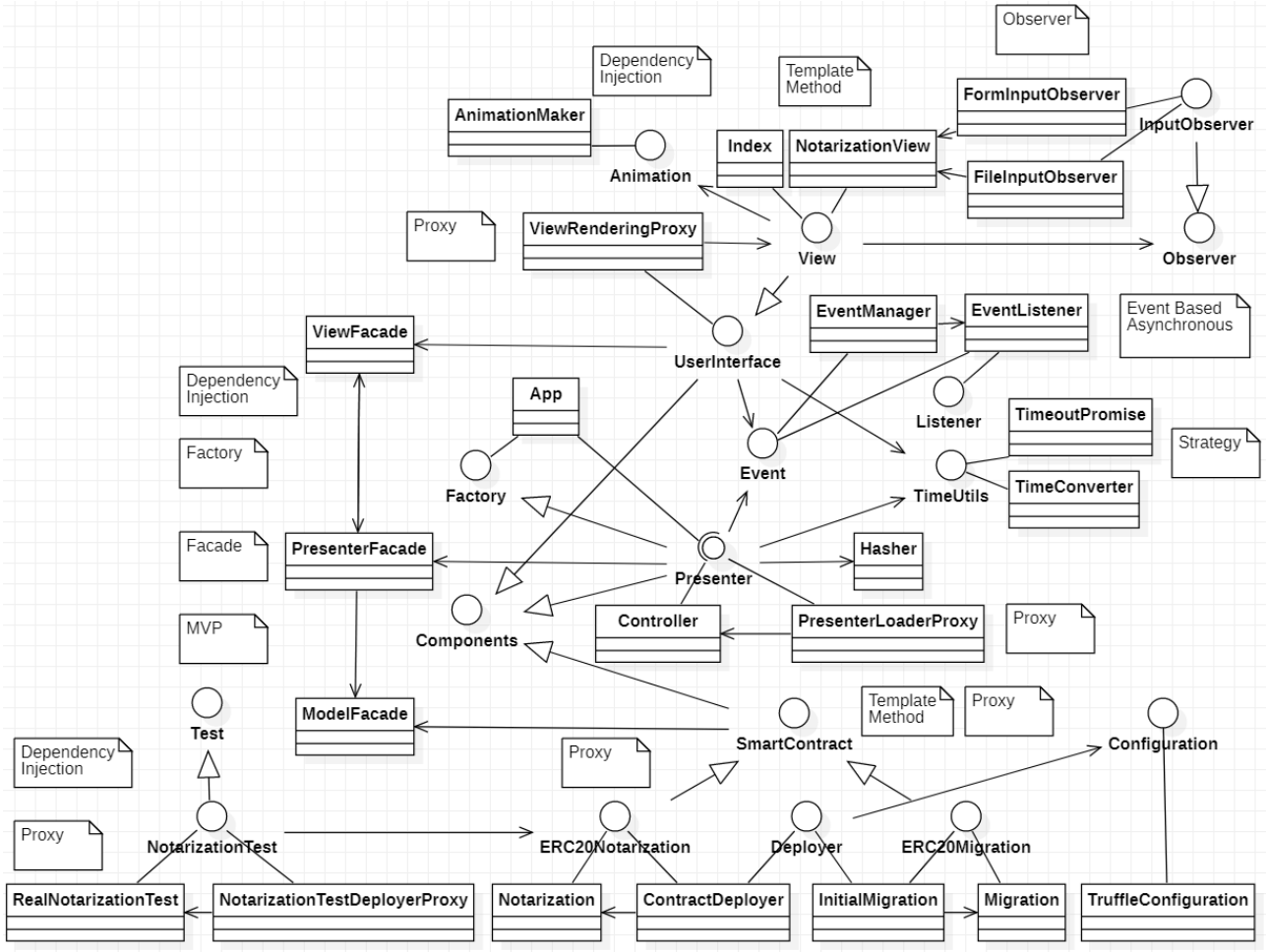

Figura 3.25: Diagramma delle classi

- **Attore primario:** un generico cliente del prodotto.
- **Attore secondario:** il *cripto wallet*.
- **Precondizioni:** l'utente ha impostato correttamente:
  1. Il sistema esterno.
  2. Il progetto e avviato l'applicazione.
- **Postcondizioni:** l'utente ha *notarizzato* e monitorato *NFT*.
- **Scenario principale:**
  1. La classe *App*, ricavata dall'interfaccia *Factory* per creare oggetti, crea la classe *Controller* a partire dall'interfaccia *Presenter*, *che* definisce la *Business Logic* e a sua volta derivata dall'interfaccia *Components* per rappresentare le caratteristiche di un generico componente del design pattern *MVP*, e utilizza la classe

PresenterLoaderProxy per il suo caricamento. Vengono quindi utilizzati i design pattern *MPV*, *Factory*, *Proxy* e *Dependency Injection*.

2. *PresenterFacade* crea:
    - Le classi *Notarization* e *Migration*, caricate rispettivamente dalle classi *ContractDeployer* e *InitialMigration,* sono ricavate dall'interfaccia *Deployer* per caricare classi su *Blockchain* e ottenute rispettivamente dalle interfacce *ERC20Notarization* ed *ERC20Migration*, entrambe derivate dall'interfaccia *SmartContract*, ricavata da *Components e* rappresentante i contratti usati dalle piattaforme decentralizzate *Web3.0*. Vengono quindi utilizzate i *design pattern Template Method* e *Proxy*. *ERC20* è il nome dello standard della piattaforma decentralizzata *Web3.0* per definire la struttura dei contratti.
    - Le classi *Index* e *NotarizationView*, renderizzate dalla classe *ViewRenderingProxy* e ottenute dall'interfaccia *View*, ricavata dall'interfaccia *UserInterface* per rappresentare l'interfaccia utente e ottenuta da Components. Vengono quindi utilizzati i *design pattern Template Method* e *Dependency Injection*.
3. *ViewFacade* comunica anche con tutto ciò che è rappresentato da *UserInterface*. *PresenterFacade* comunica anche con tutto ciò che è rappresentato da *Presenter*. *ModelFacade* comunica anche con tutto ciò che è rappresentato da *SmartContract*.
4. La comunicazione tra le componenti del *design pattern MVP* avviene dalla classe *PresenterFacade* alle classi *ViewFacade* e *ModelFacade*, e da *ViewFacade* a *PresenterFacade*, che riuniscono tutte le funzionalità di queste componenti per semplificarne l'interazione. Vengono quindi utilizzati i *design pattern Facade* e *Dependency Injection*.
5. *View* utilizza l'interfaccia *Observer*, messo a disposizione da *JavaScript* per osservare cambiamenti di stato in elementi dell'interfaccia utente, da cui si ricava l'interfaccia *InputObserver* per osservare gli elementi d'*input*, e successivamente le classi *FormInputObserver* e *FileInputObserver*, che osservano entrambe *NotarizationView*. Vengono quindi utilizzati i *design pattern Observer* e *Dependency Injection*.
6. Per disegnare animazioni grafiche nell'interfaccia utente*, View* usa la classe *AnimationMaker*, ricavata dall'interfaccia *Animation* ideata per questo scopo.
7. *Presenter* usa la classe *TimeConverter*, mentre *UserInterface* utilizza anche *TimeoutPromise*, entrambe ricavate dall'interfaccia *TimeUtils* per gestire conversioni temporali. Viene quindi utilizzato il design pattern *Strategy*.
8. L'interfaccia *Event* per comporre eventi, crea le classi *EventManager*, usato da *Presenter* e *UserInterface,* e *EventListerner*, che implementa anche l'interfaccia *Listener* per configurare un metodo per rilevare eventi. Viene quindi utilizzato il *design pattern Event Based Asynchronous*.
9. *Presenter* utilizza la classe *Hasher* per calcolare il *Hash* dei documenti appartenenti agli *NFT* da *notarizzare*.
10. Per il collegamento alle *Blockchain*, *Deployer* utilizza la classe *TruffleConfiguration*, ottenuto dall'interfaccia *Configuration* ideata per questo scopo.
11. La classe *RealNotarizationTest*, caricata dalla classe *NotarizationTestDeployerProxy*, viene ricavata dall'interfaccia *NotarizationTest* per testare *SmartContract* e derivata

dall'interfaccia *Test*, predisposta da *Node.js* per permette di creare classi di *test*. Vengono quindi utilizzati i *design pattern Proxy* e *Dependency Injection*.

3.4.3 Diagramma di sequenza

Il seguente diagramma di sequenza invece, ricavato dal diagramma delle classi e anch'esso poi descritto e spiegato approfonditamente, riassume la sequenza delle interazioni del sistema:

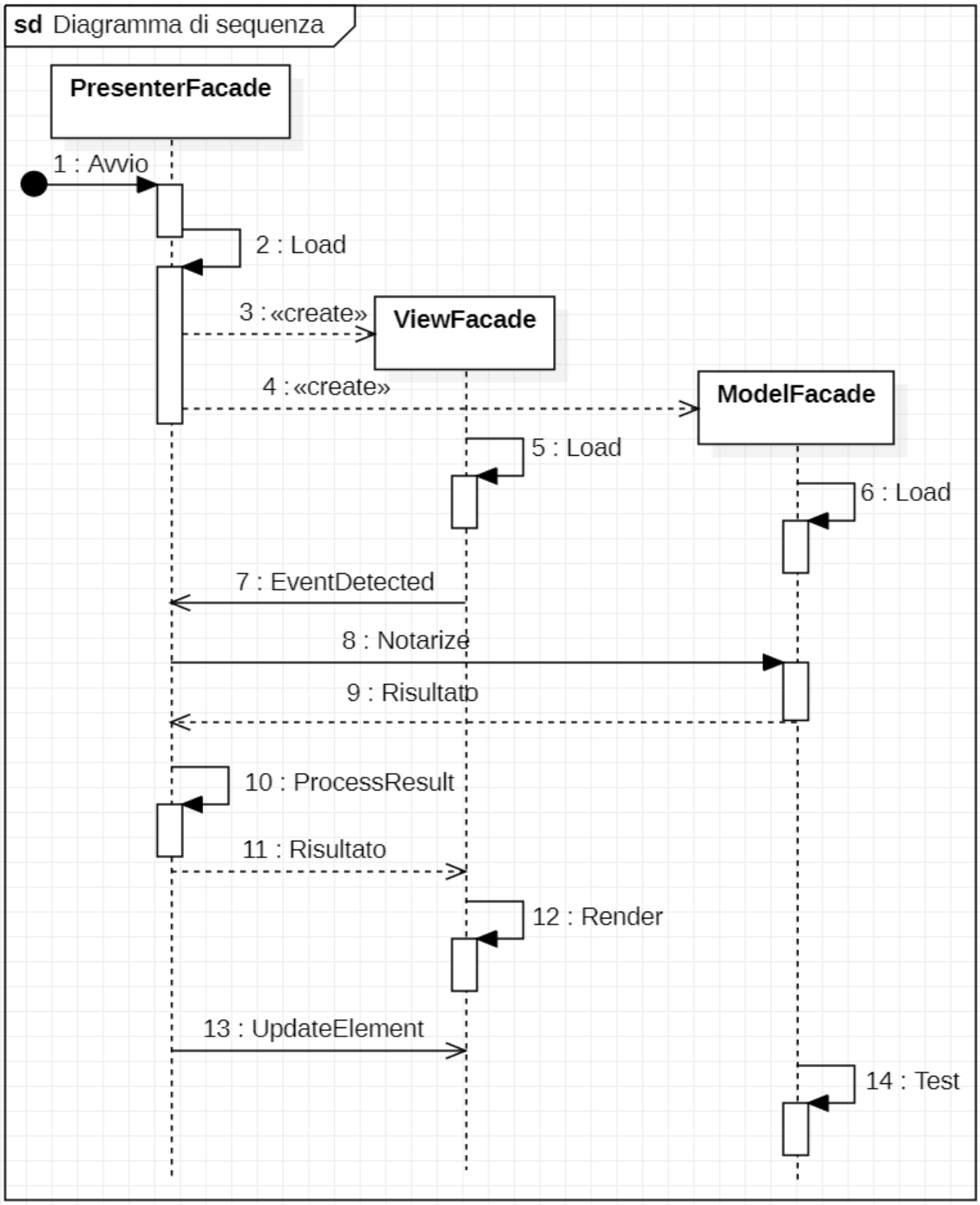

Figura 3.26: Diagramma di sequenza

- **Attore primario:** un generico cliente del prodotto.
- **Attore secondario:** il *cripto wallet*.
- **Precondizioni:** l'utente ha impostato correttamente:
    1. Il sistema esterno.
    2. Il progetto e avviato l'applicazione.
    3. Vuole *notarizzare* e monitorare *NFT*.
- **Postcondizioni:** l'utente ha *notarizzato* e monitorato *NFT*.
- **Scenario principale:**
    1. L'utente avvia il programma, innescando *PresenterFacade*.
    2. *PresenterFacade* carica le sue componenti e poi crea *ViewFacade* e *ModelFacade*, che a loro volta caricano le proprie componenti.
    3. l sistema è ora pronto a fornire il servizio, in attesa ad esempio che venga rilevato un evento nella grafica per cui *ViewFacade* deve avvertire *PresenterFacade*.
    4. Il *PresenterFacade* comanda il *ModelFacade* di eseguire la relativa operazione di *notarizzazione*.
    5. Una volta ottenuto il risultato, *PresenterFacade* lo processa e fornisce a *ViewFacade* indicazioni per visualizzare il risultato, che elabora e mostra una risposta graficamente adatta all'utente.
    6. Altra azione nasce direttamente in *PresenterFacade* a seguito del rilevamento di un evento nella grafica: in questo caso viene ripetuto il punto 5.
    7. Un'ultima azione avviene esclusivamente in *ModelFacade*, che esegue lo *Smart Contract* in modalità test se richiesto direttamente dall'utente e ne visualizza l'esito sul terminale.

3.5 Tecnologie usate

3.5.1 Linguaggi

I seguenti linguaggi di programmazione sono stati usati per lo sviluppo dell'applicazione:

- ***HTML*:** per definire la struttura dell'interfaccia utente.

Figura 3.27: Logo HTML

- ***CSS*:** per definire lo stile dell'interfaccia utente.

Figura 3.28: Logo CSS

- ***JavaScript*:** per definire il comportamento dell'interfaccia utente.

Figura 3.29: Logo JavaScript

- ***Bootstrap*:** per usare stili preconfigurati degli elementi delle pagine utente.

Figura 3.30: Logo Bootstrap

- ***Solidity*:** per sviluppare *Smart Contracts*.

Figura 3.31: Logo Solidity

- ***Web3.js*:** libreria per comunicare con *Smart Contracts*.

Figura 3.32: Logo Web3.js

- ***Batch*:** per creare lo *script* che crea l'eseguibile dell'applicazione.

Figura 3.33: Logo Batch

- ***Node.js*:** per avviare il servizio, gestire le librerie e interpretare i linguaggi di programmazione *client-side*.

Figura 3.34: Logo Node.js

3.5.2 Tecnologie

Le seguenti tecnologie sono state usate per lo sviluppo del progetto:

- ***Truffle Suite*:** libreria per gestire l'ambiente di sviluppo di *Smart Contracts*.

Figura 3.35: Logo Truffle Suite

- ***Ganache*:** piattaforma decentralizzata *Web3.0* locale per esecuzione locale e di test.

Figura 3.36: Logo Ganache

- ***Ethereum*:** piattaforma decentralizzata *Web3.0* per esecuzione in modalità reale.

Figura 3.37: Logo Ethereum

- ***IPFS*:** protocollo della libreria *Web3.js* per gestire la comunicazione con *Smart Contracts*.

Figura 3.38: Logo IPFS

- ***Lite-server*:** pacchetto di *Node.js* per gestire autonomamente l'esecuzione del programma.

Figura 3.39: Logo Lite-server

- ***Polygon*:** *Blockchain* principale del sistema.

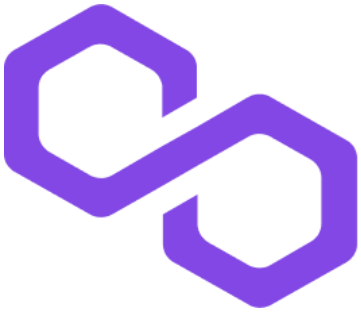

Figura 3.40: Logo Polygon

- ***Infura*:** *provider* per l'accesso alla *Blockchain*.

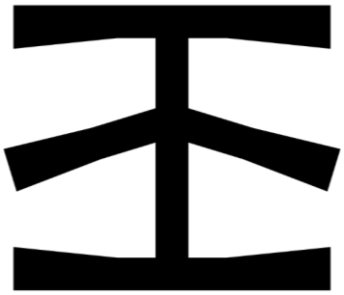

Figura 3.41: Logo Infura

- ***MetaMask*:** estensione per *browser* per la gestione del *cripto wallet*.

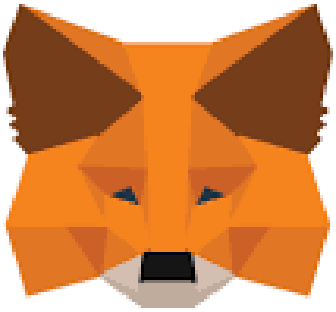

Figura 3.42: Logo MetaMask

3.6 Organizzazione della cartella di lavoro

Vale sicuramente la pena presentare l'organizzazione della cartella di lavoro, utile per l'evolvibilità futura dell'infrastruttura. La struttura della cartella di lavoro è stata creata e viene gestita da *Node.js*, in quanto esegue l'applicazione, e rispetta le indicazioni fornite da *Truffle Suite*, dato che gestisce lo sviluppo degli *Smart Contracts*. La cartella contiene i seguenti elementi, tra cui librerie usate, componenti necessari all'esecuzione del programma e file che definiscono l'applicazione:

- ***contracts*:** contiene gli *Smart Contracts*, in particolare:
  - ***Migration*:** creato automaticamente assieme al progetto. Serve alla piattaforma decentralizzata *Web3.0* per tracciare i dettagli degli altri *Smart Contracts*, quali:
    - Proprietario.
    - Ultimo caricamento.
- ***migrations*:** contiene i seguenti *script* generati automaticamente da *Truffle Suite* alla creazione del progetto:
  - ***1_initial_migration*:** carica lo *Smart Contract Migration* sulla piattaforma decentralizzata *Web3.0*.
  - ***2_deploy_contracts*:** carica gli altri *Smart Contracts*, in questo caso solo *Notarization*, sulla piattaforma decentralizzata *Web3.0*.
- ***src*:** contiene il *front-end*, in particolare:
  - Le seguenti pagine *HTML* visualizzate dall'utente:
    - ***Index*:** pagina di presentazione dell'applicazione.
    - ***Notarization*:** pagina di *notarizzazione*.
  - ***assets*:** raccoglie tutti gli elementi usati dalle pagine dell'applicazione, cioè:
    - ***css*:** contiene il file *style* che definisce lo stile delle pagine dell'applicazione.
    - ***js*:** contiene tutti gli *script* del *front-end*, in particolare:
      - ***app*:** contiene la *Business Logic* e la *User Interface*.
      - ***main*:** contiene funzioni per renderizzare la grafica.
    - ***img*:** contiene tutte le immagini usate dall'applicazione.
    - ***vendor*:** contiene le seguenti librerie usate per il *front-end*, che sono tutte relative alle pagine utente:
      - ***aos*:** contiene una collezione di animazioni per gli *scroll* verticali.
      - ***bootstrap*:** contiene un insieme di effetti, stili ed elementi grafici.
      - ***bootstrap-icons*:** contiene un insieme di icone.
      - ***glightbox*:** contiene uno *script* per visualizzare gallerie d'immagini.
      - ***isotope-layout*:** contiene una collezione di *layout* attribuibili agli elementi delle pagine dell'applicazione.
      - ***purecounter*:** contiene uno *script* per la rappresentazione di animazioni di scritte.
      - ***remixicon*:** contiene un insieme d'icone.
      - ***swiper*:** contiene uno *script* per la rappresentazione di elementi per gli *scroll* orizzontali.
- ***test*:** contiene lo *script Notarization.test* per eseguire il *back-end* in modalità di *test*.
- ***node_modules*:** contiene i moduli usati da *Node.js* per avviare l'applicazione.
- ***build*:** contiene i compilati degli *Smart Contracts*, necessari al sistema per l'esecuzione.

- ***documentation*:** contiene tutta la documentazione del progetto, ovvero questa tesi e la documentazione del progetto richiesto dall'azienda.
- Alla radice della cartella di lavoro sono presenti i seguenti file:
  - ***.gitattributes*:** contiene le caratteristiche della cartella di lavoro. È facoltativo per il programma ma necessario per il *repository* su *GitHub*.
  - ***.gitignore*:** contiene la lista degli elementi da non *versionare* sul *repository* su *GitHub*. È facoltativo sia per il programma che per il *repository* su *GitHub*.
  - ***.secret*:** se non esiste, deve essere creata. Contiene le seguenti informazioni personali dell'utente:
    - ***Mnemonic crypto wallet*:** codice segreto del proprio *crypto wallet*.
    - **Chiave nodo *Blockchain*:** chiave generata dal *provider*.
  - ***AlphaNotary*:** contiene il collegamento all'eseguibile dell'applicazione. È facoltativo sia per il progetto che per il *repository* su *GitHub*.
  - ***app*:** contiene l'eseguibile dell'applicazione. È facoltativo sia per il progetto che per il *repository* su *GitHub*.
  - ***bs-config*:** contiene la struttura della cartella.
  - ***package-lock*:** contiene il compilato della cartella.
  - ***package*:** contiene informazioni generiche e dipendenze da installare al momento della configurazione del programma.
  - ***README*:** contiene la presentazione del progetto nel *repository* su *GitHub. È* facoltativo sia per il programma che per il *repository* su *GitHub*.
  - ***LICENSE*:** contiene la licenza per il *repository* su *GitHub*. È facoltativo sia per il progetto che per il *repository* su *GitHub*.
  - ***truffle-config*:** contiene la configurazione di tutte le *Blockchain* supportate. Qui viene importata la libreria *hdwallet-provider* di *Truffle Suite* per la comunicazione con le piattaforme decentralizzate *Web3.0*.

Segue la rappresentazione della radice della cartella di lavoro:

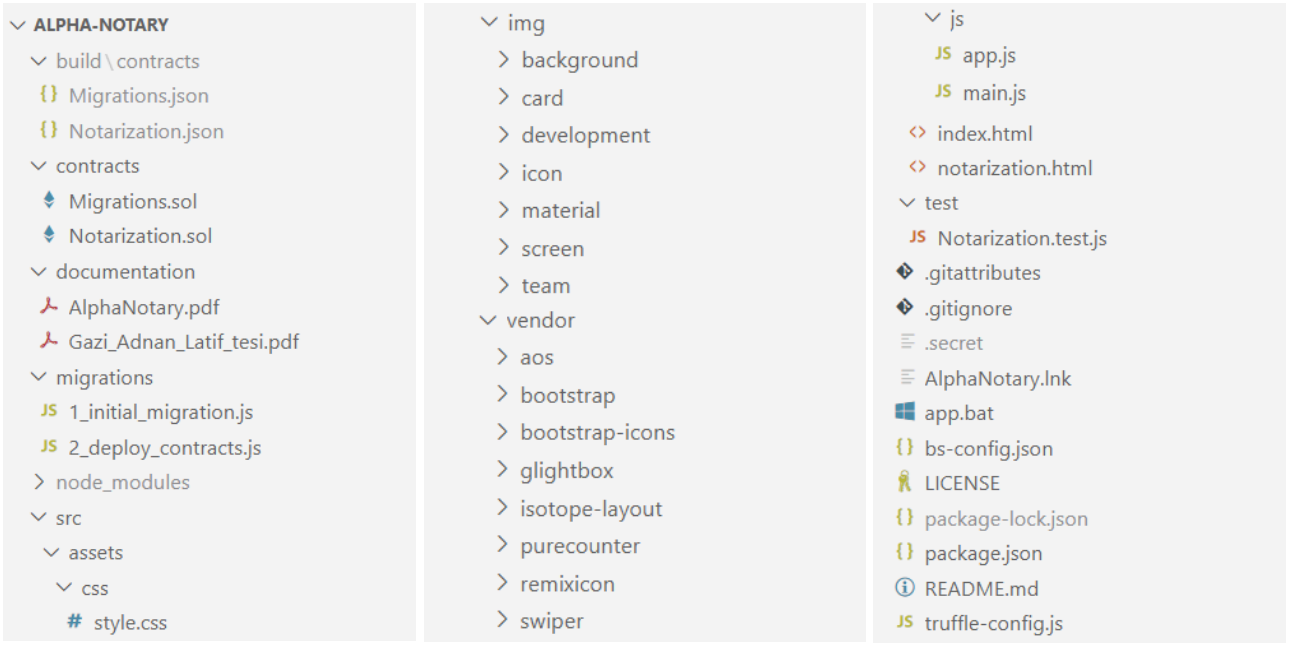

Figura 3.43: Organizzazione cartella di lavoro

4 Realizzazione

4.1 Back-end

4.1.1 Configurazione Blockchain

Come mostrato nella figura A.1 dell'appendice, vengono innanzitutto recuperate le informazioni segrete dell'utente dall'apposito *file*, e poi usate assieme ai valori delle *Blockchain* che si intende configurare per creare una lista di reti, in cui ogni elemento rappresenta la configurazione per una specifica *Blockchain* amministrata dalla libreria *hdwallet-provider*.

4.1.2 Script di Migrazione

Gli *script* di migrazione sono uno per contratto, quindi due in totale. Hanno il medesimo codice ma cambia ovviamente il contratto importato e migrato. Nella figura A.2 dell'appendice viene mostrato lo *script* relativo alla migrazione del contratto *Migration*.

4.1.3 Smart Contracts

Nello *Smart Contract* relativo alla *notarizzazione* viene usato la struttura dati mostrata nella figura A.3 dell'appendice, apposita per contenere la lista degli attributi associato ad ogni documento gestito. La lista dei documenti viene invece gestita da una tabella *hash*, che fa corrispondere la chiave di un documento al documento stesso, come mostrato nella figura A.4 dell'appendice. Il linguaggio di programmazione permette di creare funzioni per caricare dati nella *Blockchain* attualmente in uso. Nella figura A.5 dell'appendice viene mostrata la funzione che permette il caricamento degli attributi di documenti. Il resto del contratto è composto dalle funzionalità principali, ovvero quelle di *notarizzazione*. Nella figura A.6 dell'appendice viene mostrata la funzione di caricamento, che, ricevuto tutti gli attributi di un documento, li usa per crearne uno nuovo e caricarlo nella *Blockchain* e nella lista dei documenti gestiti. Poi aggiorna le interazioni, necessario per il monitoraggio.

4.1.4 Test Smart Contract Notarization

Nonostante la validazione umana del codice, l'esecuzione in modalità di *test* è necessaria per verificare l'appropriata configurazione dell'ambiente d'esecuzione e il corretto funzionamento del sistema. L'esecuzione in modalità di *test* non richiede alcuna interazione dell'utente, in quanto il programma automatizza tutte quelle necessarie tramite l'utilizzo dei *test* di unità. L'esecuzione in modalità di *test*, effettua tutte le funzionalità principali dell'applicazione, ovvero quelle relative alle operazioni di *notarizzazione*, sulla rete di prova per evitare rallentamenti e costi di transazione, e ne visualizza i risultati sottoforma di lista di *test* passati. I *test* sono divisi in quattro sezioni, quello iniziale per il caricamento degli *Smart Contracts* e poi uno per ogni funzionalità principale. Nella figura A.7 dell'appendice viene mostrato il metodo che verifica la funzione di caricamento.

4.2 Front-end

4.2.1 Business Logic

La chiave è una stringa alfanumerica non reversibile. Ciò significa che ogni documento, seppur minimamente diverso da altri, produce un risultato completamente differente. Una chiave rappresenta in modo univoco un solo documento e un documento è associato ad una sola chiave. A

partire dal documento chiunque può ottenere la sua chiave, in quanto questa procedura è pubblica, ma a partire dalla chiave nessuno è in grado di ricavare il documento originale. È anche opportuno citare che la complessità algoritmica del codice è sempre al più lineare, rendendo il sistema algoritmicamente quanto più efficiente possibile. Ne rimane comunque una complessità computazionale molto elevata a causa di un'infrastruttua interna e esterna molto pesante.

Una delle funzioni più interessanti nella *Business Logic* è relativa alla generazione di una chiave a partire da un documento mediante l'operazione di *hash*. Nelle figure A.8 e A.9 dell'appendice viene mostrata tale funzione. Essa è composta dalle seguenti parti:

- **Lettura:** legge il documento e sfrutta l'algoritmo *SHA-256* fornito dalla libreria *crypto* per ricavarne il valore *hash*. È in grado di gestire più documenti contemporaneamente, in questo caso genera un unico codice *hash* per l'insieme dei documenti, ed eventuali errori durante il processo. Al termine chiama la seconda parte.
- **Conversione:** converte il risultato in esadecimale scorrendolo *byte* per *byte*, ne aggiusta la lunghezza e restituisce il risultato finale.

Un'altra funzione importante riguarda la connessione con l'infrastruttura esterna e il caricamento delle componenti del sistema. Essa, come mostrato nella figura A.10 dell'appendice, è suddivisa nei seguenti passi:

1. **Caricamento componenti**: carica le componenti secondo l'ordine progettato.
2. **Importazione *crypto wallet*:** ottiene dal *browser* la *crypto wallet* utilizzata e lo usa per connettersi a *MetaMask*. Notifica l'errore altrimenti.
3. **Connessione *Blockchain*:** effettua una transazione a vuoto per verificare il funzionamento dell'infrastruttura esterna. Notifica l'errore altrimenti.
4. **Configurazione profilo utente**: ottiene il saldo del profilo dalla *crypto wallet* utilizzata e lo visualizza nell'interfaccia utente. Notifica l'errore altrimenti.
5. **Importazione *Smart Contracts*:** imposta i contratti per poterne usare le funzioni.

Il resto della *Business Logic* è composto dalle funzionalità principali, ovvero quelle di *notarizzazione*. Di seguito viene mostrato la funzione di caricamento. Le altre funzioni sono presentano la medesima struttura delle operazioni. Segue il codice in questione nelle figure A.11 e A.12 dell'appendice. Essa è formata dalle seguenti operazioni:

1. Acquisizione delle informazioni immesse dall'utente nell'interfaccia grafica come richiesti dall'operazione in questione e controllo del corretto formato.
2. Generazione di tutti gli altri attributi necessari, come ad esempio il valore *hash* del documento, alla relativa funzione nello *Smart Contract*.
3. Esecuzione della relativa funzione di *notarizzazione* nel contratto.
4. Visualizzazione dei risultati nell'interfaccia utente e pulizia dei campi utilizzati per l'inserimento dei dati.
5. Aggiunta dell'interazione appena eseguita per la sua visualizzazione nella sezione resoconto operazioni, sempre se attivata la modalità completa.
6. Gestione di qualunque eventuale errore con conseguente notifica di un messaggio apposito nell'interfaccia utente.

4.2.2 User Interface

Per quanto concerne invece l'interfaccia grafica, una sua parte fondamentale è l'inizializzazione degli eventi da cui partono le catene di interazioni interne al sistema. Nella figura A.13 dell'appendice viene mostrato un esempio di come ciò vengano impostati, in questo caso per il caricamento di documenti nella dedicata area dell'interfaccia utente. Seguono poi, nelle figure A.14 e A.15 dell'appendice, due tra le più importanti funzioni del *front-end*, rispettivamente per l'animazione durante il caricamento dei documenti e la loro visualizzazione dopo il caricamento, per mostrare come queste funzioni uniscano la definizione della logica alla creazione della grafica. È infine doveroso sottolineare come l'interfaccia utente sia completamente accessibile, in quanto *responsive*, ovvero adattabile ad ogni tipologia di dispositivo, ottimizzato per *SEO* (*Search Engine Optimization*), con pagine di dimensioni ridotte per permetterne un veloce caricamento e con codice valido come confermato dai servizi automatici *online* per la validazione di codice di relativo alla struttura di pagine utente.

5 Prodotto finale

5.1 Esecuzione del sistema

5.1.1 Requisiti per l'avvio

Prima di tutto sarà necessario avere i seguenti sistemi configurati sul proprio dispositivo:

- ***Infura*:** bisogna creare un proprio profilo nel sito *web* del servizio, avviare un abbonamento, gratuito se sotto una determinata soglia giornaliera di blocchi con cui sono state effettuate interazioni, sufficienti per lo sviluppo del sistema ma non per l'uso professionale dell'applicazione, impostare le *Blockchain* che si intende utilizzare, in questo caso tutte, e importare per ognuna di esse l'indirizzo di rete fornito su *MetaMask*.
- ***MetaMask*:** è necessario installare l'estensione per il *browser* con cui si intende usare l'applicazione e configurare il proprio *wallet*, creando il proprio profilo, accreditando un ammontare sufficiente di *criptovaluta* per usufruire dell'infrastruttura esterna e importando le reti di *Blockchain*.
- ***Truffle suite*:** bisogna installare la libreria eseguendo il seguente comando sul terminale:

```
npm install -g truffle
```

Figura 5.1: Comando installazione Truffle Suite

La libreria sarà stata installata correttamente se eseguendo il seguente comando viene mostrato la lista dei linguaggi di programmazione, con le relative versioni, installati per i progetti *Truffle Suite*:

```
truffle version
```

Figura 5.2: Comando verifica installazione Truffle Suite

- ***Node.js*:** è necessario scaricare il *widget* d'installazione con cui installarlo. La tecnologia sarà stata installata correttamente se eseguendo il seguente comando viene mostrato la versione attualmente presente nel dispositivo:

```
node
```

Figura 5.3: Comando verifica installazione Node.js

- ***Ganache*:** è necessario scaricare il *widget* d'installazione con cui installarlo. Successivamente bisogna avviare l'applicazione *desktop* e creare una propria piattaforma decentralizzata *Web3.0*. Ciò serve solamente per eseguire l'applicazione in locale, in cui *Ganache* deve essere sempre in esecuzione, ed è utile durante la fase di sviluppo o se si intende usare il sistema solamente in locale, evitando rallentamenti e costi di transazione. L'esecuzione locale del servizio non è obbligatoria.

5.1.2 Installazione della cartella di lavoro

Bisogna disporre dell'intera cartella di lavoro in locale. Se non lo si ha già, può essere recuperata dal *cloud* dell'impresa o dal *server* aziendale, oppure scaricando o clonando il *repository* su *GitHub* Successivamente sarà necessario aprire la cartella da terminale ed eseguire i seguenti comandi:

```
truffle compile
npm install
```

Figura 5.4: Comandi installazione cartella di lavoro

Il primo comando esegue il *back-end* per compilare gli *Smart Contracts* e produrre la cartella *build*. Viene anche prodotto il *file package-lock* assieme. Il secondo comando invece, crea la cartella *node_modules* e qui vi installa tutti pacchetti di dipendenze del sistema.

5.1.4 Avvio del servizio

Per avviare il servizio, bisogna aprire il progetto da terminale ed eseguire i seguenti comandi:

```
truffle compile [--network <your_network>] [--reset]
npm start
```

Figura 5.5: Comandi esecuzione del servizio

Il secondo comando avvia il sistema. Il primo comando esegue la migrazione degli *Smart Contracts* nella *Blockchain*, ed è da eseguire ad ogni modifica dei contratti. Il comando comprende i seguenti paramentri opzionali:

- ***--network*:** è seguito dal nome della *Blockchain* in cui si intende eseguire la migrazione tra quelli configurati nel *file truffle-config*. Se omesso, effettua la migrazione nella rete locale.
- ***--reset*:** cancella la precedente copia, se esistono, di ogni contratto in caricamento, perdendo quindi tutti i relativi dati d'utilizzo. Se omesso, carica gli *Smart Contract* se non presenti già, altrimenti li aggiorna e mantiene i loro dati d'uso.

Seguono le immagini più rilevanti dell'applicazione in esecuzione e che ne mostrano l'utilizzo delle funzionalità, ovvero quelle di *notarizzazione*:

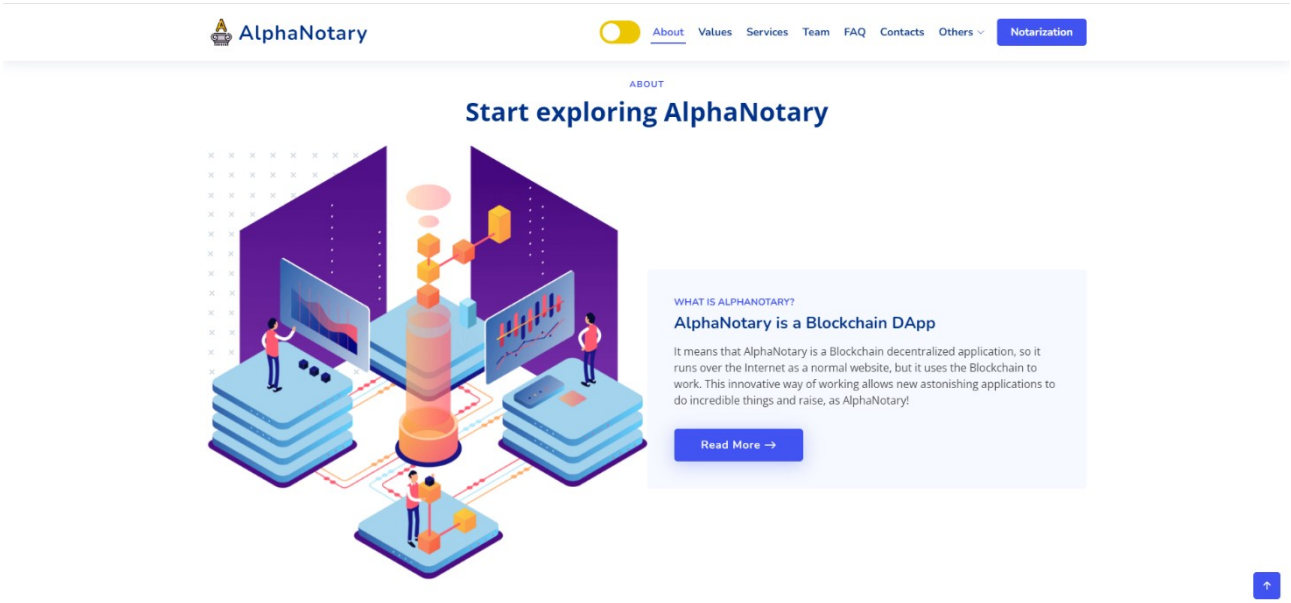

Figura 5.6: Screen di una sezione della pagina di presentazione dell'applicazione

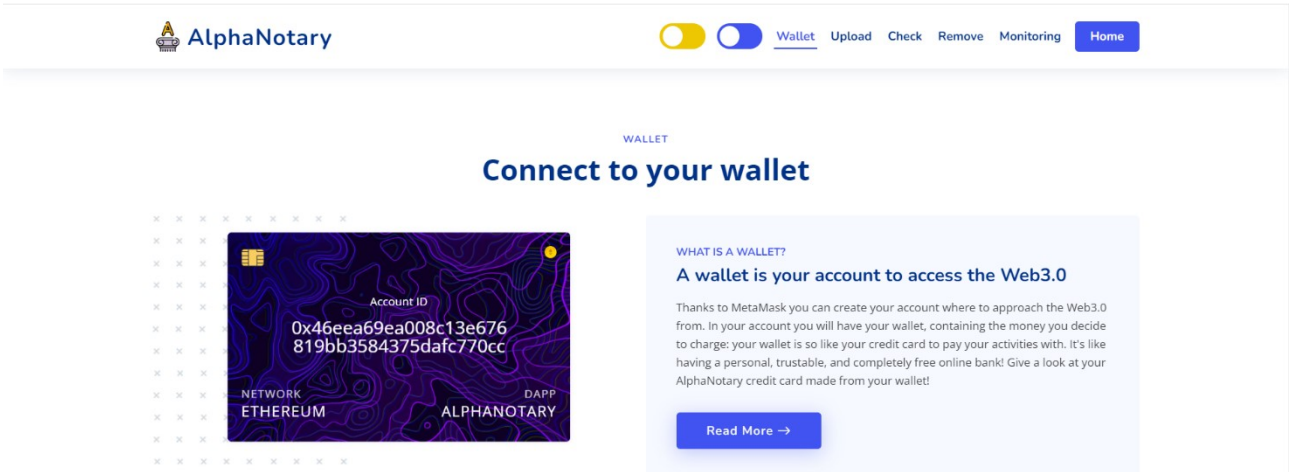

Figura 5.7: Screen della sezione di visualizzazione del proprio wallet dell'applicazione

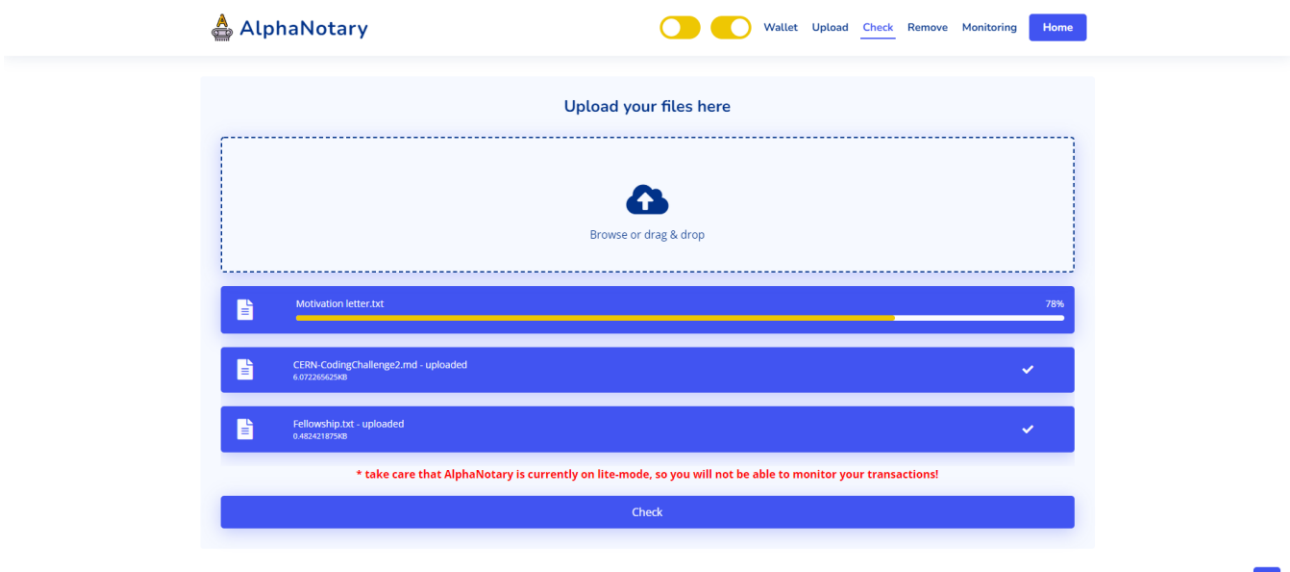

Figura 5.8: Screen della sezione di caricamento dell'applicazione

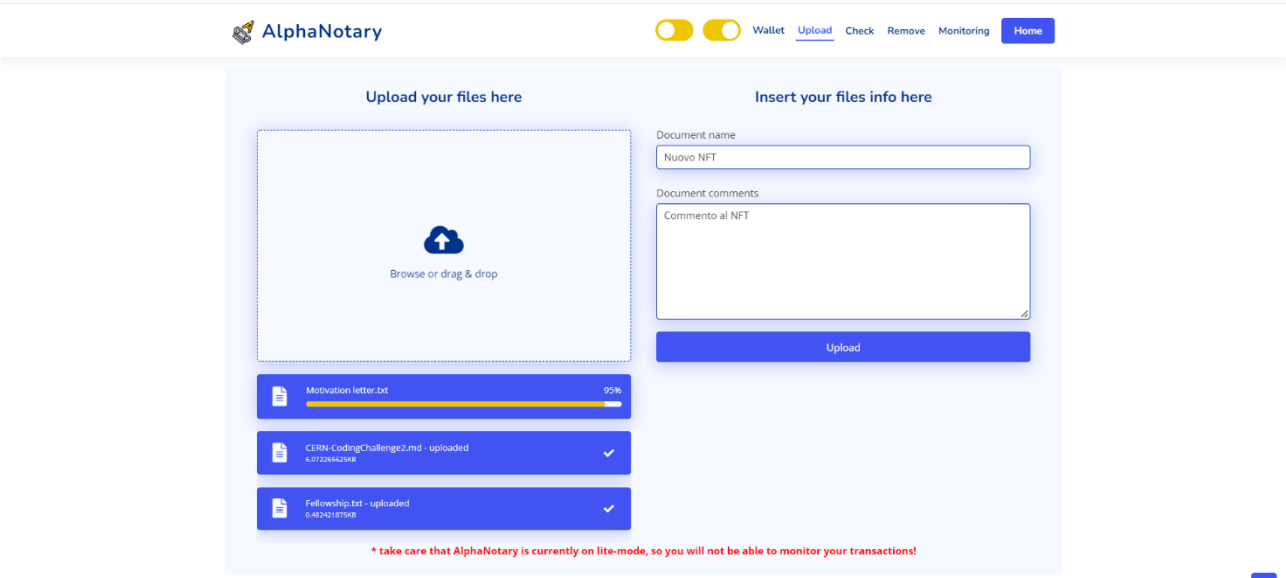

Figura 5.9: Screen della sezione di autenticazione o rimozione dell'applicazione

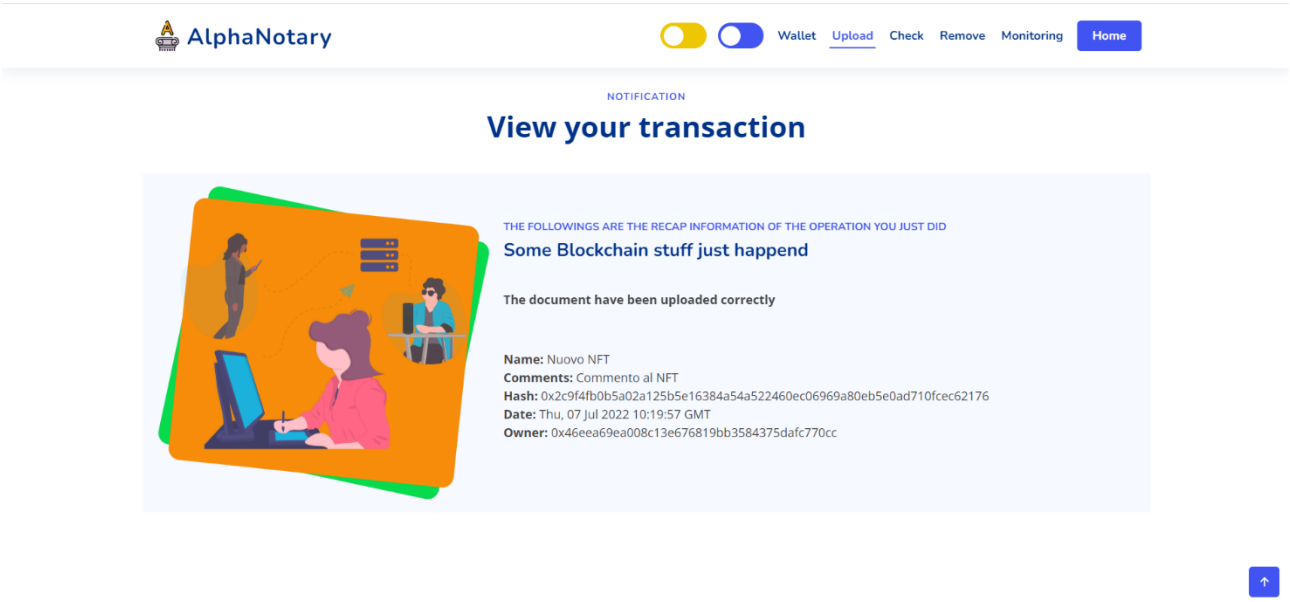

Figura 5.10: Screen della sezione di visualizzazione esito dell'applicazione

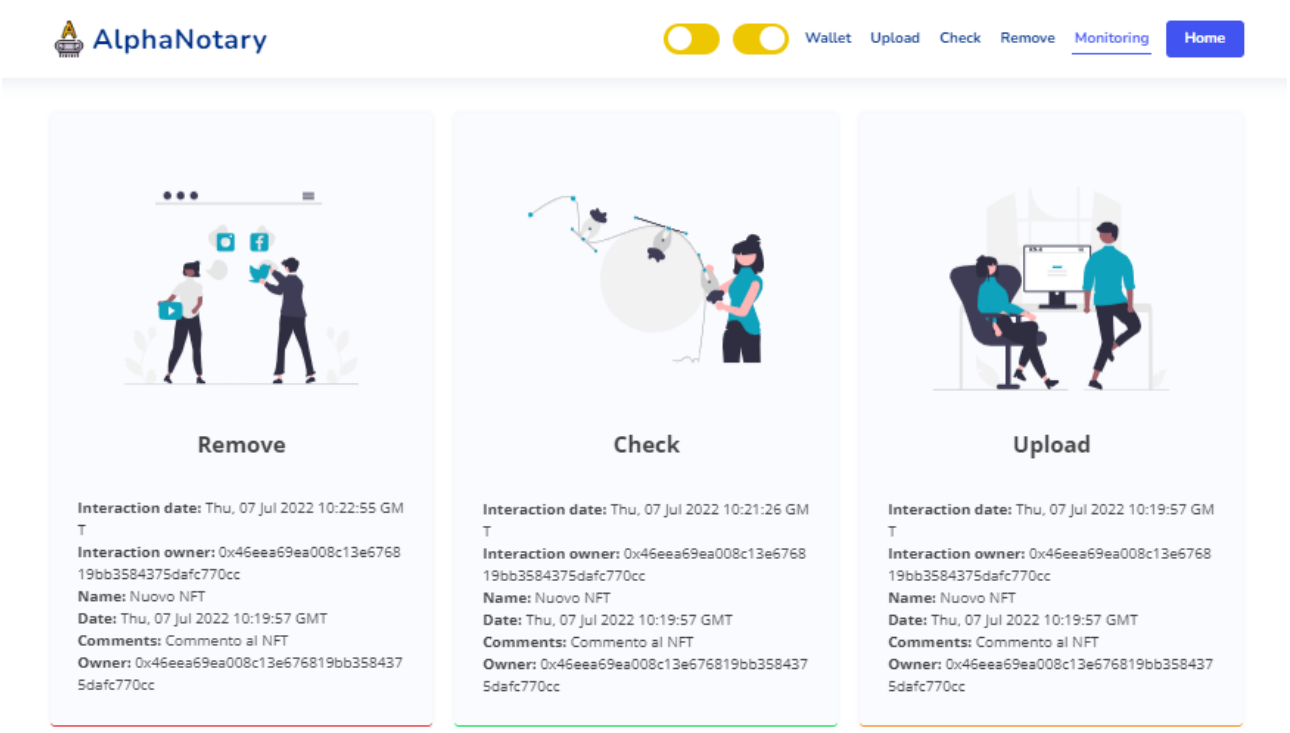

Figura 5.11: Screen della sezione di resoconto operazioni dell'applicazione

5.1.3 Verifica e validazione del codice

Per avviare il programma in modalità di *test*, bisogna aprire il progetto da terminale ed eseguire i seguenti comandi:

```
truffle test
```

Figura 5.12: Comando esecuzione modalità di test

Se l'ambiente d'esecuzione e la cartella di lavoro saranno stati configurati correttamente, verrà visualizzato il seguente risultato:

```
Using network 'development'.


Compiling your contracts...
===========================
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\Notarization.sol
> Artifacts written to C:\Users\adnan\AppData\Local\Temp\test--10964-D8lcBjegTPKU
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang


  Contract: NOTARIZATION
    ✓ DEPLOY (231ms)
    ✓ UPLOAD (4116ms)
    ✓ CHECK (1909ms)
    ✓ REMOVE (2550ms)


  4 passing (9s)
```

Figura 5.13: Risultato esecuzione in modalità di test

5.2 Costo delle transazioni

Essendo che l'applicazione interagisce frequentemente con le *Blockchain* e ogni interazione ha delle spese e un tempo d'attesa, alcune volte diventa proibitivo utilizzare il sistema a causa di costi che arrivano all'ordine delle decine di dollari e tempi che raggiungono i minuti d'attesa per operazione.

Il sistema è stato quindi costruito quanto più efficiente possibile in modo da migliorare le prestazioni, alleviando questa problematica. Inoltre, è stata anche sviluppata una funzionalità per cambiare dinamicamente *Smart Contract*, passando da quello completo, che offre tutte le funzionalità principali quali le operazioni di *notarizzazione* e quelle secondarie, a quello leggero, che mantiene tutte le funzionalità principali disabilitando però la funzione di monitoraggio, e viceversa. L'intento è attenuare il problema effettuando un numero minore di azioni nella *Blockchain* e utilizzare un contratto che richiedere una minore quantità di dati per avere transazioni più leggere.

È stato analizzato solo il costo delle transazioni, in quanto il tempo d'attesa non è monitorato dalla *crypto wallet* utilizzata. L'esperimento è stato ripetuto tre volte per avere un risultato più accurato, calcolato appunto dalla loro media. Ogni volta sono state eseguite tutte le operazioni del sistema che utilizzano le *Blockchain* in tutte le modalità previste.

L'esperimento sarebbe stato sfalsato se non fosse stato svolto eseguendo il programma in locale, in quanto decisamente più stabile delle reti reali sia a livello di spese che tempi d'attesa per ogni azione. Infatti, quest'ultime sono ad ogni istante molto variabili da questo punto di vista, a causa del numero di *miners* a disposizione della piattaforma decentralizzata *Web3.0* in ogni momento, i quali ne determinano la potenza computazionale totale offerta. Di seguito è riportata la tabella dei risultati dell'esperimento:

Costo transazioni					
Test	**Lite mode**	**Operazione**	**Costo (ETH)**	**Costo Medio (ETH)**	**Differenza**
#1	No	Upload	0,00082326	0,00185042	77%
	No	Check	0,00054228		
	No	Remove	0,00048488		
	Sì	Upload	0,00015754	0,00042755	
	Sì	Check	0,00011982		
	Sì	Remove	0,00015019		
#2	No	Upload	0,00597366	0,01503830	75%
	No	Check	0,00429796		
	No	Remove	0,00476668		
	Sì	Upload	0,00158410	0,00381620	
	Sì	Check	0,00092370		
	Sì	Remove	0,00130840		
#3	No	Upload	0,00614142	0,01512158	75%
	No	Check	0,00429772		
	No	Remove	0,00468244		
	Sì	Upload	0,00158386	0,00381548	
	Sì	Check	0,00092346		
	Sì	Remove	0,00130816		
				Differenza totale	**75%**

Figura 5.14: Tabella dei risultati dell'esperimento relativo ai costi delle transazioni

Si conclude quindi che la modalità leggera diminuisca i costi delle transazioni di un notevole 75%, il che rappresenta una differenza colossale, soprattutto considerando la situazione a lungo termine, in cui si prevede un uso intensivo del sistema con molteplici transazioni ad ogni istante e utilizzando una *Blockchain* reale, in cui i costi delle transazioni sono generalmente più elevati rispetto ad una rete di prova. I vantaggi apportati dalla funzionalità per cambiare dinamicamente *Smart Contract* superano di gran lunga l'onere di lavoro per svilupparla, giustificandone la scelta di realizzarla.

5.3 Distribuzione del servizio

5.3.1 Server aziendale

Come richiesto, è stato innanzitutto prodotto un pacchetto contenente tutto il lavoro svolto durante il tirocinio e caricato sul *cloud* dell'impresa. Successivamente, l'seguibile del programma è stato caricato e avviato sul *server* aziendale. È stato possibile effettuare questa procedura grazie ad *Apache HTTP Server*, che ha permesso di installare la cartella di lavoro e avviare il servizio come sopra descritto. Essendo un server, l'assenza dell'interfaccia utente ha costretto a svolgere tutte le operazioni da terminale.

5.3.2 Online

Infine, il prodotto è stato anche caricato in un *repository, su GitHub* (https://github.com/adnangazi/alpha-notary), condiviso e soggetto a *Continuous Integration/Continuous Delivery* (*CI/CD*) mediante l'utilizzo di *Vercel*. Però, la porta su cui il sistema viene ascoltato dal *server* è attualmente gestita in modo automatico, e non è nemmeno possibile impostarla esplicitamente. Oltretutto, essendo che *Vercel* non riesce a identificare la porta se non espressamente dichiarata, nonostante l'applicazione venga caricata correttamente, non si riesce ad ascoltare il servizio, ma è solo possibile, conoscendo il percorso della cartella di lavoro, risalire alle pagine *HTML* che costituiscono l'interfaccia utente.

5.4 Evoluzioni future dell'infrastruttura

5.4.1 Mantenimento

Il progetto è *open source*, quindi rilasciato con una licenza di diritto d'autore che concede agli utenti i diritti di utilizzare, studiare, modificare e distribuire il *software* e il relativo codice sorgente a chiunque e per qualsiasi scopo. In particolare, per contribuire, sia che sia per mantenere o per sviluppare il sistema, è necessario lavorare ad una copia del codice sorgente del *repository* su *GitHub*, dove bisognerà poi effettuare la richiesta di caricamento delle modifiche. Se esse verranno accettate da un amministratore di sistema, andranno automaticamente in produzione, come impostato dal meccanismo di *Continuous Integration/Continuous Delivery* (*CI/CD*).

5.4.2 Sviluppo

Segue la lista dei possibili sviluppi futuri del sistema:

- **Versione *Express* dell'applicazione:** necessario per la distribuzione del sistema su *Vercel*, in quanto permette una gestione più approfondita della configurazione dell'applicazione, tra cui l'impostazione della porta su cui ascoltare il servizio.
- **Automatizzazione delle transazioni:** necessario per creare un *API* del sistema e comodo per evitare ad ogni operazione di *notarizzazione* di dover confermare l'avvio della transazione.
- ***API* del sistema:** necessario per integrare il sistema agli altri prodotti aziendali.
- ***Framework* per il *front-end* (*Angular*):** comodo se dovesse aumentare la complessità del *front-end*. Attualmente la semplicità del sistema non ne costringe l'uso. È stato scelto *Angular* per allinearsi con le tecnologie usate dall'azienda e per la popolarità di questo linguaggio di programmazione, che ne garantisce notevole documentazione e supporto.
- **Architettura a *microservizi*:** comodo per facilitare l'integrazione di funzionalità al prodotto in esecuzione senza interrompere il servizio, ad aumentare la modularità delle componenti e per facilitare quindi la creazione di un *API* del sistema più facilmente e più velocemente.
- **Gestione di profili e abbonamenti utenti:** consigliabile se si intende commercializzare il prodotto, rendendolo quindi a pagamento.

6 Conclusioni sull'esperienza

6.1 Consuntivo di processi e fasi

Il lavoro è stato organizzato in modo accurato, infatti, il consuntivo, calcolato sommando le ore di lavoro dal dettaglio delle attività relative ai seguenti processi, è stato così ripartito a livello di ore di lavoro per processo:

Durata in ore	Descrizione dell'attività
60 (-5)	**Formazione sui domini coinvolti nello sviluppo del sistema**
70 (+5)	**Progettazione e documentazione**
15 (+5)	*Analisi del problema e del dominio applicativo*
15 (-5)	*Progettazione della soluzione e relativi test*
10 (+5)	*Pianificazione del lavoro*
30	*Documentazione sulla progettazione, realizzazione e presentazione*
160 (+10)	**Realizzazione dei sistemi e implementazione delle funzionalità**
80 (+5)	*Realizzazione dei sistemi*
80 (+5)	*Implementazione delle funzionalità*
30 (-5)	**Verifica, collaudo e dimostrazione**
15 (-5)	*Verifica della progettazione e del codice*
10	*Collaudo dei sistemi realizzati e delle funzionalità implementate*
5	*Dimostrazione del sistema finale agli stakeholders*
Totale ore	**320**

Figura 6.1: Consuntivo tabella ripartizione delle ore di lavoro per processo

Ogni valore riporta accanto lo scostamento rispetto al preventivo e, nonostante molti di essi siano differenti dall'originale, gli scostamenti sono talmente sottili da non rappresentare differenze significative nella realizzazione del lavoro rispetto a quanto organizzato. Quindi, visto che il preventivo e il consuntivo coincidono grossomodo, le fasi di lavoro e le attività al dettaglio, e di conseguenza anche il rispettivo diagramma di *Gantt*, sono rimaste invariate. Il consuntivo, calcolato invece dalle ore di lavoro dal dettaglio delle attività relative ad ogni fase di lavoro, è stato ripartito a livello temporale per fase di lavoro come mostrato dal seguente diagramma di *Gantt*:

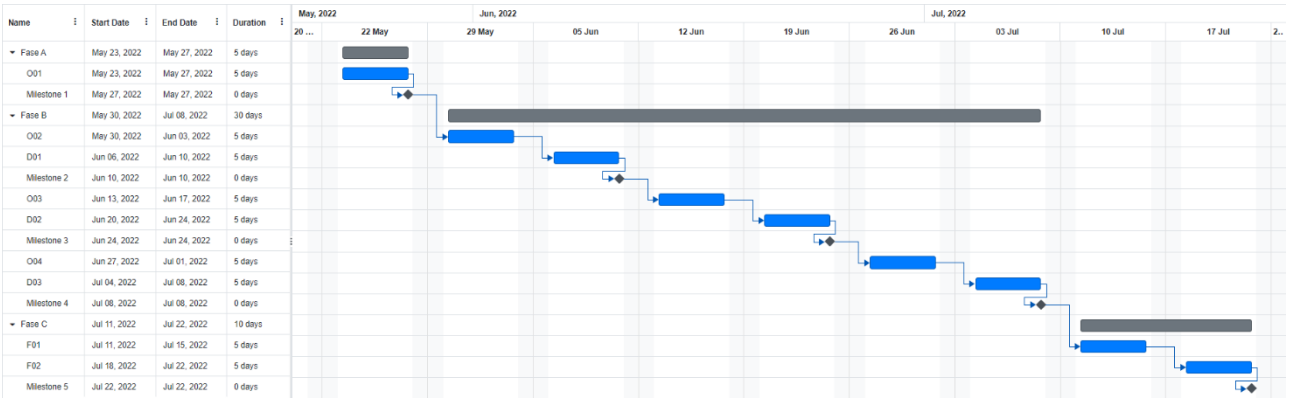

Figura 6.2: Consuntivo diagramma di Gantt ripartizione del tempo per le fasi di lavoro

6.2 Raggiungimento dei vincoli

6.2.1 Realizzazione dei requisiti

Come si può evincere dalla progettazione dell'architettura del sistema, accertare dalla sua realizzazione e infine verificare dal prodotto finale, sono stati realizzati tutti i requisiti specifici del progetto, e quindi anche quelli inizialmente imposti dall'azienda. L'adempimento dei requisiti obbligatori assicura che il risultato finale sia conforme alle richieste del tirocinio, mentre quelli desiderabili ne garantiscono la completezza al fine di aggiungerne maggior valore, e infine quelli opzionali determinano la qualità del risultato. Come in questo caso, il pieno conseguimento di tutti questi fattori garantiscono un risultato e una valutazione ottimale.

6.2.2 Conseguimento degli obiettivi

Un progetto di tale calibro sarebbe impossibile da completare in assenza di una iniziale formazione dello studente sui domini coinvolti nel progetto. Inoltre, l'esito del tirocinio non preclude la possibilità di future collaborazioni. Infine, vista la realizzazione dei requisiti e che alcuni di questi prevedevano la redazione di documentazione, queste serviranno successivamente per l'espansione di questo prodotto e facilitare la comprensione, e quindi la realizzazione, di quelli analoghi, consentendo all'azienda di proseguire lo sviluppo del sistema ed espandere l'uso delle *Blockchain* in molteplici contesti d'uso. Il successo del progetto indica quindi che anche gli obiettivi del progetto sono stati pienamente conseguiti.

6.3 Conoscenze acquisite

6.3.1 Abilità tecniche

Quest'esperienza ha permesso di acquisire molteplici nuove conoscenze e competenze tecniche relative al dominio applicativo del progetto, ovvero le *Blockchain* e tutti gli strumenti, *software*, linguaggi di programmazione e tecnologie ad esso annessi. Grazie a questo stage è infatti possibile, anche partendo da conoscenze e competenze minime insegnate dal Corso di Laurea, diventare operativi in questo ambito in modo da potervi lavorare autonomamente. Inoltre, sono state consolidate le abilità già possedute in merito ai processi di progettazione, documentazione, realizzazione, gestione dell'ambiente di lavoro, verifica e collaudo, mentre quelle relative al processo di distribuzione, sono state espanse.

6.3.2 Competenze trasversali

Un progetto di questo livello induce a sviluppare competenze trasversali quali capacità organizzative, risolutive di problemi e di veloce apprendimento, in quanto, in loro assenza, sarebbe impossibile raggiungere l'obiettivo prefissato. Inoltre, affinché l'esito del tirocinio sia positivo, è necessario sviluppare una buona capacità d'integrazione in un nuovo ambiente, quale quello lavorativo. Il successo del progetto indica quindi che queste competenze trasversali sono state pienamente conseguite.

6.4 Valutazione del lavoro

6.4.1 Considerazioni sul progetto

L'applicazione ha una rilevante utilità in quanto offre funzionalità che apportano grossi benefici, come ad esempio quello di autenticazione, altamente richieste e utilizzate laddove è necessario gestire una grande mole di documenti, soprattutto legali, come in enti giuridici o apparati burocratici. Oltretutto, non esistono molti altri sistemi in commercio che forniscono questo stesso servizio. Le *Blockchain* sono una delle tecnologie in maggiore ascesa nell'ultimo periodo, e in tanti prospettano che in un prossimo futuro saranno alla base di molti sistemi, nonché al centro di studi, sviluppi e divulgazioni, come già succede in alcune nicchie. La chiave di questo successo risiede nel concetto rivoluzionario di *Blockchain* stessa, grazie a cui è possibile implementare sistemi con caratteristiche uniche come *AlphaNotary*, ai quali, unito alla fama di questa tecnologia, viene garantito forte notorietà. Questi sono i motivi per cui questo prodotto sarebbe in grado di ottenere grande successo se venisse commercializzato.

È necessario però ribadire che le *Blockchain* presentano anche numerosi difetti, come ad esempio il costo d'utilizzo, i tempi di completamento delle operazioni e l'instabilità del servizio. È quindi consigliabile sviluppare applicazioni del genere non per convenienza legata alla notorietà di questa tecnologia, ma solamente se necessari i suoi pregi, e se questi superano le sue debolezze. Il dominio applicativo decisamente interessante unito all'onere richiesto dall'applicazione sia a livello di dimensioni del lavoro che difficoltà delle attività, rendono senz'altro il progetto un ottimo esercizio per acquisire nuove conoscenze e competenze, nonché per mettersi alla prova. Infine, il progetto risulta impegnativo ma fattibile da completare con le conoscenze fornite dal Corso di Laurea ed entro i tempi previsti dal tirocinio.

6.4.2 Giudizio sullo stage

L'esperienza di stage è decisamente positiva: infatti, non solo il progetto è stato completato in modo ottimale, ma in generale l'azienda garantisce l'inserimento dello studente nel mondo del lavoro con successo, il tutor ha fornito l'aiuto essenziale per concludere le attività con esito positivo ma senza privare lo studente del proprio merito, le modalità di lavoro incentivano la qualità delle mansioni eseguite e l'ambiente aziendale è tale da costruire profondi legami d'amicizia con i colleghi. Quindi, sono state pienamente soddisfatte le aspettative in merito all'esperienza sia per lo studente, sia per il Corso di Laurea che per l'azienda.

Appendice

Sono di seguito rappresentate le figure relative a codice:

```
const HDWalletProvider = require('@truffle/hdwallet-provider');
const fs = require('fs');
const lines = fs.readFileSync(".secret", 'utf-8').split('\n');
const mnemonic = lines[0].trim();
const infuraKey = lines[1];

module.exports = {
  networks: {
    polygon: {
      provider: () => new HDWalletProvider(mnemonic, `https://polygon-mainnet.
      infura.io/v3/${infuraKey}`),
      network_id: 137
    },
```

Figura A.1: Codice configurazione Blockchain

```
const Migrations = artifacts.require("Migrations");

/**
* deploy the Migration Smart Contract
*
* @param deployer Truffle interface for API that deploys to Blockchain
*/
module.exports = function(deployer) {
  deployer.deploy(Migrations);
};
```

Figura A.2: Codice script di migrazione

```
/**
* Document object containing all the information wanted to save for each
document notarized
*/
struct Document {
    string name;
    uint256 date;
    string comments;
    address owner;
    bool isSet;
}
```

Figura A.3: Codice struttura dati della lista attributi documenti

```
/**
* hash map list of Documents
*
* @ param number that stands for the hash code
* @ return Document object in the position of the hash code mapped
*/
mapping(uint256 => Document) private documentMapping;
```

Figura A.4: Codice tabella hash per gestione lista di documenti

```
/**
* event to emit a Document change to the Blockchain
*
* @param _name string of the name of the Document
* @param _date number of the date of the initial upload
* @param _comments string of the comments of the Document
* @param _owner address of the owner of the Interaction
* @param _isSet bool to identify if a Document is set correctly
*/
event documentEntry(string _name, uint256 _date, string _comments, address
indexed _owner, bool _isSet);
```

Figura A.5: Codice caricamento dati relativo a un documento nella Blockchain

```
/**
* upload or update a Document, and eventually record the Interaction
*
* @param _name string of the name of the Document
* @param _hashValue number of the hash code of the Document
* @param _comments string of the comments of the Document
* @param fullMode bool to identify if to record the Interaction
*/
function upload(string memory _name, uint256 _hashValue, string memory
_comments, bool fullMode) public {
    uploadResult = documentMapping[_hashValue].isSet;
    Document memory document = createDocument(_name, _comments);
    documentMapping[_hashValue] = document;
    emitDocument(document);
    setCurrentDocument(document);

    if (fullMode) {
        Interaction memory interaction = createInteraction(uploadResult ?
        "Update" : "Upload", _hashValue);
        interactionMapping[interactionCount++] = interaction;
        emitInteraction(interaction);
    }
}
```

Figura A.6: Codice funzione di caricamento in Smart Contract

```
/**
* test the upload functionality
*/
it("UPLOAD", async () => {
  // uploading a Document
  await this.notarization.upload(this.documentName, this.documentHash, this.documentComments, true);

  // testing the current Document information
  var attualDate = new Date(0);
  assert(attualDate.setUTCSeconds(await this.notarization.getCurrentDocumentDate()) <= Date.now());
  assert.equal(await this.notarization.getCurrentDocumentOwner(), this.owner);
  assert.equal(await this.notarization.getCurrentDocumentName(), this.documentName);
  assert.equal(await this.notarization.getCurrentDocumentComments(), this.documentComments);
  assert.equal(await this.notarization.getUploadResult(), false);

  // testing the current Document Interaction
  attualDate = new Date(0);
  assert(attualDate.setUTCSeconds(await this.notarization.getCurrentInteractionDate()) <= Date.now());

  // testing the current Interaction
  var interaction = await this.notarization.getInteractionInfo(await this.notarization.getInteractionCount() - 1);
  assert.equal(interaction[0], "Upload");
  attualDate = new Date(0);
  assert(attualDate.setUTCSeconds(interaction[1]) <= Date.now());
  assert.equal(interaction[2], this.owner);
  assert.equal(interaction[3], this.documentName);
  attualDate = new Date(0);
  assert(attualDate.setUTCSeconds(interaction[4]) <= Date.now());
  assert.equal(interaction[5], this.documentComments);
  assert.equal(interaction[6], this.owner);
  assert.equal(interaction[7], true);
});
```

Figura A.7: Codice test Smart Contract Notarization

```
/**
* create a hash code of files
*
* @param files list of files
* @return string of the hash code
*/
createHash(files) {
  /**
  * correct the format of the hash code and convert to hex
  *
  * @param buffer string of the previous hash code
  * @return string of the correct hash code
  */
  function toHex(buffer) {
    var padding;
    var hexCodes = [];
    var view = new DataView(buffer);

    // correcting format
    for (var i = 0, n = view.byteLength, k = Uint32Array.BYTES_PER_ELEMENT; i <
    n; i += k) {
      padding = '00000000';
      hexCodes.push((padding + (view.getUint32(i)).toString(16)).slice(-padding.
      length));
    }

    // correcting to hex
    var hash = hexCodes.join('');

    if (hash.substring(0, 2) !== '0x') {
      hash = "0x" + hash;
    }

    return hash;
  }
```

Figura A.8: Codice creazione hash 1

```
  /**
  * convert the files to a hash code
  *
  * @param resolve object of the result
  * @param reject object of the error
  * @return string of the hash code
  */
  return new Promise(function (resolve, reject) {
    var reader = new FileReader();

    /**
    * encrypt the files to a hash code
    */
    reader.onload = function () {
      crypto.subtle.digest('SHA-256', this.result)
        /**
        * manage the correction of the format
        *
        * @param hash string of the hash code
        */
        .then(function (hash) {
          resolve(toHex(hash));
        })
        /**
        * manage the error
        *
        * @param err string of the error
        */
        .catch(function (err) {
          console.error(err);
        });
    };

    reader.onerror = reject;
    reader.readAsArrayBuffer(new Blob(files));
  });
}
```

Figura A.9: Codice creazione hash 2

```
/**
* load components
*/
load: async () => {
  View.eventLoader();

  // importing MetaMask
  if (typeof web3 !== 'undefined') {
    Controller.web3Provider = web3.currentProvider;
    web3 = new Web3(web3.currentProvider);
  } else {
    View.bannerNotify(["Please connect to Metamask!"], 0);
  }

  // importing Web3
  if (window.ethereum) {
    window.web3 = new Web3(ethereum);

    try {
      await ethereum.enable();
      web3.eth.sendTransaction({/* ... */ });
    } catch (error) {
      console.log('Non-Ethereum Blockchain operations ongoing...');
    }
  } else if (window.web3) {
    Controller.web3Provider = web3.currentProvider;
    window.web3 = new Web3(web3.currentProvider);
    web3.eth.sendTransaction({/* ... */ });
  } else {
    View.bannerNotify(["Non-Ethereum browser detected. You should consider
    trying MetaMask!"], 0);
  }
  // importing account
  Controller.account = web3.eth.accounts[0];

  if (Controller.account != undefined) {
    document.getElementById('account').innerHTML = Controller.account;
  } else {
    View.bannerNotify(["Error getting account ID: MetMask account is not
    connected!"], 0);
  }

  View.importWallet();

  // importing Smart Contract
  var Notarization = TruffleContract(await $.getJSON('Notarization.json'));
  Notarization.setProvider(Controller.web3Provider);
  Controller.notarization = await Notarization.deployed();
  web3.eth.defaultAccount = web3.eth.accounts[0];
  View.monitoring();
},
```

Figura A.10: Codice caricamento Business Logic

```
/**
* upload in the Blockchain a file
*/
uploadDocument: async () => {
  // checking the correct format of the values for the operation
  var name = View.otherValues[3].value;
  var emptyName = name == "";
  var majorName = name.lastIndexOf('>');
  var minorName = name.indexOf('<');
  var correctFormatName = majorName != -1 && minorName != -1 ? majorName <
  minorName : true;

  if (View.files[0].length > 0 && !emptyName && correctFormatName) {
    Utils.createHash(View.files[0])
      /**
      * upload in the Blockchain a file
      *
      * @param hash string of the hash code of the file
      */
      .then(async (hash) => {
        // getting other information
        var comments = View.otherValues[4].value;
        var majorComments = comments.lastIndexOf('>');
        var minorComments = comments.indexOf('<');

        if (majorComments != -1 && minorComments != -1 ? majorComments <
        minorComments : true) {
          await Controller.notarization.upload(name, hash, comments, Controller.
          contractDecisor);
```

Figura A.11: Codice funzione di caricamento Business Logic 1

```
            // visualizing in the GUI the result
            View.bannerNotify(["The document have been " + (await Controller.
            notarization.getUploadResult() ? "updated " : "uploaded ") +
            "correctly!", "Name", name, "Comments", comments, "Hash", hash,
            "Date", Utils.epochConverter(await Controller.notarization.
            getCurrentDocumentDate()), "Owner", await Controller.notarization.
            getCurrentDocumentOwner()], 1);

            // resetting the uploading stuff
            View.resetLine(View.otherValues[3]);
            View.resetLine(View.otherValues[4]);
            View.resetArea(View.progressAreas[0]);
            View.resetArea(View.uploadedAreas[0]);
            View.resetFile(0);

            // inserting the current Interaction and updating the wallet base
            if (Controller.contractDecisor) {
              View.importWallet();
              View.insertInteraction(await Controller.notarization.
              getInteractionCount() - 1);
            }
          } else {
            View.bannerNotify(["There may be an HTML tag in the comments!"], 1);
          }
        })
        /**
        * manage the error
        *
        * @param err string of the error
        */
        .catch(function (err) {
          console.error(err);
        });
```

Figura A.12: Codice funzione di caricamento Business Logic 2

```
/**
* on drag over event load
*/
View.forms[0].addEventListener("dragover", async (event) => {
  event.preventDefault();
  View.openDrag(View.forms[0], View.dragTexts[0]);
});
```

Figura A.13: Codice funzione di caricamento User Interface

```
/**
* create a paragraph in a GUI element with a particular format for the strings
to notify
*
* @param containerLines element where to insert the paragraph
* @param toNotify list of string to insert in the paragraph
* @param deleyer number of the milliseconds to sleep before the animations
(default 0)
*/
insertLines: async (containerLines, toNotify, deleyer = 0) => {
  var amount = 50;
  var deley = deleyer + amount;

  // paragraph title
  containerLines.innerHTML = '<p data-aos="fade-up" data-aos-delay="' + deley +
  '"><b>' + toNotify[0] + '</b></p>';
  deley += amount;

  // inserting like in JSON format
  for (var i = 1; i < toNotify.length; i++) {
    if (i % 2) {
      containerLines.innerHTML += '</br><span data-aos="fade-up"
      data-aos-delay="' + deley + '"><b>' + toNotify[i] + ':</b> </span>';
    } else {
      containerLines.innerHTML += '<span data-aos="fade-up" data-aos-delay="' +
      deley + '">' + toNotify[i] + '</span>';
      deley += amount;
    }
  }
},
```

Figura A.14: Codice funzione di visualizzazione contenuto nell'interfaccia utente

```
/**
* insert in the GUI the uploaded file
*
* @param files list of files
* @param files list of files
*/
function uploadedLoading(file, num) {
  // calculating the file dimension
  let dim = "";
  let size = file.size / 1024;

  if (size <= 1024) {
    dim = size + "KB";
  } else {
    size /= 1024;
    dim = size + "MB";
  }

  /**
  * sleep and insert the uploaded file
  */
  Utils.sleep(2200).then(function () {
    View.progressAreas[num].innerHTML = '';
    View.uploadedAreas[num].innerHTML += '<li class="row"><div
    class="content"><i class="fas fa-file-alt"></i><div class="details"><span
    class="name">' + file.name + ' - uploaded</span><span class="size">' +
    dim + '</span></div></div><i class="fas fa-check"></i></li>';
  });
}
```

Figura A.15: Codice funzione animazione caricatamente documenti

Glossario

Acronimi, abbreviazioni, parole ambigue, meno comuni e più caratteristiche del progetto sono di seguito approfonditamente descritte:

- ***Blockchain*:** letteralmente catena di blocchi, è una struttura dati condivisa e immutabile, definita come un registro digitale le cui voci sono raggruppate in blocchi, concatenati in ordine cronologico, e la cui integrità è garantita dall'uso della *crittografia*. Sebbene la sua dimensione sia destinata a crescere nel tempo, è immutabile nel concetto di quanto il suo contenuto una volta scritto tramite un processo normato, non è più né modificabile né eliminabile, a meno di non invalidare l'intero processo. Tali tecnologie sono incluse nella più ampia famiglia dei registri distribuiti, ossia sistemi che si basano su un registro distribuito, che può essere letto e modificato da più nodi di una rete.Non è richiesto che i nodi coinvolti conoscano l'identità reciproca o si fidino l'uno dell'altro perché, per garantire la coerenza tra le varie copie, l'aggiunta di un nuovo blocco è globalmente regolata da un protocollo condiviso. Una volta autorizzata l'aggiunta del nuovo blocco, ogni nodo aggiorna la propria copia privata. La natura stessa della struttura dati garantisce l'assenza di una sua manipolazione futura. Le caratteristiche che accomunano i sistemi sviluppati con le tecnologie della *Blockchain* e dei registri distribuiti sono digitalizzazione dei dati, decentralizzazione, disintermediazione, tracciabilità dei trasferimenti, trasparenza/verificabilità, immutabilità del registro e programmabilità dei trasferimenti. Grazie a tali caratteristiche, la *Blockchain* è considerata pertanto un'alternativa in termini di sicurezza, affidabilità, trasparenza e costi alle banche dati e ai registri gestiti in maniera centralizzata da autorità riconosciute e regolamentate, come pubbliche amministrazioni, banche, assicurazioni e intermediari di pagamento [B1].
- ***DApp*:** applicazione decentralizzata, ovvero che funziona in modo autonomo, in genere attraverso l'uso di contratti intelligenti, eseguiti su un sistema informatico decentralizzato, *Blockchain* o altro sistema di contabilità distribuita. Come le applicazioni tradizionali, forniscono alcune funzioni o utilità ai propri utenti. Tuttavia, funzionano senza l'intervento umano e non sono di proprietà di nessuna entità, ma distribuiscono *token* che rappresentano la proprietà. Questi *token* vengono distribuiti secondo un algoritmo programmato agli utenti del sistema, diluendo la proprietà e il controllo dell'applicazione. Senza un'entità che controlla il sistema, l'applicazione diventa decentralizzata. Le applicazioni decentralizzate sono state rese popolari dalle tecnologie di registro distribuito come la *Blockchain* di *Ethereum*, su cui sono costruite le *dApp*, tra le altre *Blockchain* pubbliche. La natura affidabile e trasparente delle *dApp* ha portato a maggiori sviluppi nell'utilizzo di queste funzionalità all'interno dello spazio della finanza decentralizzata. Le *dApp* sono suddivise in 17 categorie: scambi, giochi, finanza, gioco d'azzardo, sviluppo, archiviazione, ad alto rischio, portafoglio, governance, proprietà, identità, media, social, sicurezza, energia, assicurazioni e salute [B10].
- ***Smart Contract*:** protocollo informatico che facilita, verifica, o fa rispettare la negoziazione o l'esecuzione di un contratto, permettendo talvolta la parziale o la totale esclusione di una

clausola contrattuale. Di solito hanno anche un'interfaccia utente e spesso simulano la logica delle clausole contrattuali [B7].

- ***Miner:*** dispositivi che effettuano *mining*, che, nel contesto della tecnologia *Blockchain*, è il processo di aggiunta di transazioni al grande registro pubblico distribuito delle transazioni esistenti, noto come *Blockchain*. Il termine è meglio conosciuto per la sua associazione con *Bitcoin*, sebbene altre tecnologie che utilizzano le *Blockchain* utilizzino il *mining*. Il *mining* di *Bitcoin* premia le persone che eseguono operazioni di *mining* con più *Bitcoin*. Il *mining* di *Blockchain* comporta l'aggiunta di transazioni al registro *Blockchain* esistente delle transazioni distribuite tra tutti gli utenti di una *Blockchain*. Mentre il *mining* è principalmente associato al *Bitcoin*, anche altre tecnologie che utilizzano una *Blockchain* utilizzano il *mining*. Il *mining* implica la creazione di un *hash* di un blocco di transazioni che non può essere facilmente falsificato, proteggendo l'integrità dell'intera *Blockchain* senza la necessità di un sistema centrale. L'estrazione viene in genere eseguita su un computer dedicato, poiché richiede una *CPU* veloce, oltre a un maggiore consumo di elettricità e più calore generato rispetto alle normali operazioni del computer. L'incentivo principale per il *mining* è che gli utenti che scelgono di utilizzare un dispositivo per il *mining* vengono ricompensati per farlo. Nel caso di *Bitcoin*, sono 25 *Bitcoin* per *hash*. Questo è il motivo per cui alcuni *hacker* utilizzano macchine in cui si rompono per estrarre *Bitcoin*, facendo in modo che una vittima inconsapevole paghi i costi del *mining* senza trarne alcun vantaggio [B3].
- ***Ethereum:*** piattaforma decentralizzata del *Web3.0* per la creazione e pubblicazione *peer-to-peer* di contratti intelligenti creati in un linguaggio di programmazione *Turing*-completo. La *criptovaluta* a esso legata, *Ether* (*ETH*), è seconda in capitalizzazione dietro ai *Bitcoin*. Per poter girare sulla rete *peer-to-peer*, i contratti di *Ethereum* pagano l'utilizzo della sua potenza computazionale tramite una unità di conto che funge quindi sia da *criptovaluta* sia da carburante. In altre parole, contrariamente a molte altre *criptovalute*, *Ethereum* non è solo un *network* per lo scambio di valore monetario, ma una rete per far girare contratti basati su *Ethereum*. Questi contratti possono essere utilizzati in maniera sicura per eseguire un vasto numero di operazioni. *Ethereum* fa parte di un gruppo di piattaforme *Blockchain* detto di nuova generazione. Come per le altre *criptovalute*, la validità di ciascun *Ether* è garantita da una *Blockchain*, che è un elenco di record in continua crescita, chiamati blocchi, i quali sono collegati tra loro e protetti mediante *crittografia*. Per definizione, la *Blockchain* è in sé resistente alla modifica dei dati. È un libro mastro contabile, aperto e distribuito che registra le transazioni tra due parti in modo efficiente e permanentemente verificabile [B2].
- ***Cryto*****:** rappresentazione digitale di valore basata sulla *crittografia*. Si tratta di una risorsa digitale paritaria e decentralizzata. Al mondo esistono oltre 17.500 *criptovalute*. Le *criptovalute* utilizzano tecnologie di tipo *peer-to-peer* su reti i cui nodi risultano costituiti da dispositivi di utenti, situati potenzialmente in tutto il globo. Su queste apparecchiature vengono eseguiti appositi programmi che svolgono funzioni di portamonete. Non c'è attualmente alcuna autorità centrale che le controlla. Le transazioni e il rilascio avvengono collettivamente in rete, pertanto non c'è una gestione di tipo centralizzato. Queste proprietà, uniche nel loro genere, non possono essere esplicate dai sistemi di pagamento tradizionale. Il controllo decentralizzato di ciascuna *criptovaluta* funziona attraverso una tecnologia di contabilità generalizzata, in genere una *Blockchain*, che funge da *database* di transazioni finanziarie pubbliche. Un sistema decentralizzato potrebbe essere più resistente

ad attacchi informatici o a incidenti operativi rispetto a un sistema accentrato perché il primo continua a operare anche quando uno o più nodi smettono di funzionare [B8].

- ***Notarizzazione*:** processo eseguito dal notaio, ovvero il soggetto al quale è affidata la funzione di garantire la validità dei contratti, degli atti giuridici civili e dei negozi giuridici, attribuendo pubblica fede agli atti e sottoscrizioni apposte alla sua presenza [B4].
- ***Wallet*:** portafoglio elettronico, ovvero un programma o un servizio *web* che permette agli utenti di memorizzare e controllare in maniera centralizzata le proprie informazioni personali inerenti agli acquisti *online*, come *login*, *password*, indirizzi di spedizione e dettagli dei propri strumenti di pagamento come carte di credito o altri servizi di pagamento collegati a strumenti o conti bancari o di moneta elettronica. Fornisce una adeguata, rapida ed agevole tecnologia che permette all'utente di effettuare acquisti di prodotti presso qualunque persona o negozio al mondo [B5].
- ***Web3.0*:** descrive l'evoluzione dell'utilizzo del *web* e l'interazione fra gli innumerevoli percorsi evolutivi possibili. Questi includono trasformare il *web* in un *database*, cosa che faciliterebbe l'accesso ai contenuti da parte di molteplici applicazioni che non siano dei *browser*, sfruttare al meglio le tecnologie basate sull'intelligenza artificiale, il *web* semantico, il *geospatial web*, l'implementazione di una *Blockchain* distribuita che supporti un linguaggio *Turing*-completo [B11].
- ***Faucet*:** applicazione o un sito *web* che distribuisce piccole quantità di *criptovalute* come ricompensa per il completamento di attività facili. Viene loro dato il nome di rubinetti perché le ricompense sono piccole, proprio come piccole gocce d'acqua che gocciolano da un rubinetto che perde. Tuttavia, nel caso dei *criptovalute*, piccole quantità di *criptovaluta* gratuite o guadagnate vengono inviate al portafoglio di un utente. Per ottenere *criptovalute* gratuite, gli utenti devono completare attività semplici come visualizzare annunci, guardare video di prodotti, completare quiz, aprire *link* o completare un *captcha*. Non sono certamente uno schema per arricchirsi rapidamente. Più semplice è il compito, minore è la ricompensa. La maggior parte dei siti *web* offre una soglia minima di pagamento. Quindi, i premi guadagnati completando le attività vengono depositati in un portafoglio *online* del sito. Un utente può ritirare questo premio solo dopo aver raggiunto la soglia minima impostata. Con i migliori *criptovalute*, questo potrebbe richiedere solo un giorno, ma spesso può richiedere più di una settimana [B9].
- ***NFT* (*non-fungible token*):** in italiano gettone non fungibile o riproducibile, è un tipo speciale di *token* che rappresenta l'atto di proprietà e il certificato di autenticità scritto su catena di blocchi di un bene unico, digitale o fisico. I gettoni non fungibili non sono quindi reciprocamente intercambiabili. Ciò è in contrasto con le *criptovalute*, come *Bitcoin* e molti gettoni di rete o di utilità, che sono per loro stessa natura fungibili, ovvero possono essere duplicati infinite volte in copie esattamente identiche ed interscambiabili, impossibilitando dunque di definire univocamente un'identità del singolo gettone che lo differenzi da tutti gli altri e rendendo perciò tutte le copie equivalenti e identiche al gettone originale, e anche in relazioni agli usi e funzioni [B6].

Fonti

Bibliografia

Le fonti esterne provenienti da libri sono di seguito elencate, ognuna assieme al proprio simbolo associato, capitolo di riferimento, edizione, autori e produttore:

B1. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 1: What is Ethereum. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B2. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 2: Ethereum Basics. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B3. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 3: Ethereum Clients. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B4. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 4: Cryptography. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B5. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 5: Wallets. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B6. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 6: Transactions. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B7. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 7: Smart Contracts and Solidity. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B8. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 9; Smart Contracts Security. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B9. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 10: Tokens. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B10. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 12: Decentralized Applications (DApps). Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

B11. Mastering Ethereum: Building Smart Contracts and DApps - 1° edizione – capitolo 13: The Ethereum Virtual Machine. Andreas M. Antonopoulos, Dr. Gavin Wood - O'Railly Media, Inc - 1005 Gravenstein Hughway North, Sebastopol, CA 95472.

Sitografia

Le fonti esterne provenienti da siti *web* sono di seguito elencate, ognuna assieme al proprio simbolo associato e *link*:

S1. *Venicecom S.r.l.* storia: https://www.venicecom.it/chi-siamo/la-nostra-storia

S2. *Venicecom S.r.l.* impresa: https://www.venicecom.it/chi-siamo/venicecom-nel-mondo

S3. *Venicecom S.r.l.* valori:

https://www.venicecom.it/chi-siamo/i-valori-che-guidano-il-nostro-agire

S4. *Venicecom S.r.l. ProQ*: https://www.venicecom.it/prodotti/pro-q

S5. *Venicecom S.r.l. Kalmo*: https://www.venicecom.it/prodotti/kalmo

S6. *Venicecom S.r.l. KronosApp*: https://www.venicecom.it/prodotti/kronosapp

S7. Modello *Agile:* https://it.wikipedia.org/wiki/Metodologia_agile

S8. Metodologia *Scrum:* https://it.wikipedia.org/wiki/Scrum_(informatica)

S9. *Continuous Integration/Continuous Delivery* (*CI/CD*):

https://www.redhat.com/it/topics/devops/what-is-ci-cd

S10. Architettura monolitica*:* https://universeit.blog/architettura-monolitica

S11. Architettura a *layer:* https://it.wikipedia.org/wiki/Architettura_multi-tier

S12. *Design Pattern:* https://it.wikipedia.org/wiki/Design_pattern

S13. Principi di programmazione *SOLID:* https://it.wikipedia.org/wiki/SOLID