Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

PROTOTIPAZIONE DI UN FRAMEWORK WEB UTILIZZANDO IL VIRTUAL DOM CON APPROCCIO TEST DRIVEN DEVELOPMENT

PROTOTYPING WEB FRAMEWORK USING VIRTUAL DOM WITH TEST DRIVEN DEVELOPMENT

GIULIO FAGIOLI

Relatore: *Lorenzo Bettini*

Anno Accademico 2020-2021

INDICE

PREFAZIONE

Nella seguente tesi di laurea verranno introdotti alcuni dei concetti su cui si basano i moderni framework web, per comprenderne meglio il funzionamento e sfruttarli nel modo più corretto.
Per affrontare la stesura è stata necessaria la realizzazione di una versione prototipale di quest'ultimi, come framework web, implementando solamente alcune delle funzionalità comunemente supportate.
È risultato di fondamentale importanza lo studio, durante il tirocinio, del Document Object Model e delle problematiche rispetto alle prestazioni ad esso legate negli applicativi web di dimensioni medio grandi, poiché hanno permesso di potersi interrogare sull'ottimizzazione di alcune operazioni e sulle problematiche presenti.
La scelta di utilizzare il concetto di Virtual DOM è stata presa per la sua facilità di implementazione e il vasto utilizzo in molti dei moderni framework web, inoltre la vasta documentazione presente online ha permesso di affrontare e risolvere i problemi che si sono presentati durante lo sviluppo.
Il linguaggio di programmazione scelto per la realizzazione del progetto di laurea è TypeScript, un linguaggio strongly typed la cui crescita in termini di utilizzo ha subito un incremento nel corso degli anni. Si è scelto di utilizzare le metodologie di sviluppo Test Driven Development e Mutation Testing, per la realizzazione del progetto.
Queste tecniche hanno permesso di scrivere codice qualitativamente migliore, funzionante e facilmente estensibile. Proprio l'aggiunta di nuove funzionalità durante la fase di sviluppo ha beneficiato di queste metodologie, attraverso la suite di test realizzata è stato possibile rivedere e modificare più volte il funzionamento di specifiche classi e funzioni avendo la confidenza sul corretto funzionamento del framework.

La tesi si articola nei seguenti capitoli:

Il capitolo 1 elenca i concetti che sono risultati necessari per la codifica del progetto di laurea, verranno trattati il linguaggio TypeScript e le sue caratteristiche, JSX, Il DOM, il concetto di Virtual DOM con una potenziale alternativa ed il Test Driven Development come metodologia

di sviluppo.

Il capitolo 2 espone le principali caratteristiche e moduli che compongono un framework web, più nello specifico saranno descritte le scelte progettuali effettuate e le funzionalità presenti nel framework sviluppato.

Il capitolo 3 riguarda un analisi più dettagliata dei tipi, classi e funzioni implementate, al fine di descrivere i design pattern utilizzati e l'effettivo comportamento del framework, fornendo anche rappresentazioni grafiche utili per una migliore comprensione dell'algoritmo di diffing.

Il capitolo 4 elencata alcuni dei 152 test realizzati fra Unit, Integration e End to end, che hanno assicurato il corretto funzionamento del framework, inoltre fornisce un'introduzione su come elementi di pagine web o applicativi web possono essere testati.

1

CONCETTI BASE E STATO DELL'ARTE

Nel corso del seguente capitolo verranno elencati i concetti utilizzati nel progetto di laurea.
Verrà introdotto brevemente TypeScript ed alcune delle funzionalità maggiormente utilizzate verranno dettagliate, verrà esposto il Virtual DOM, un concetto attraverso il quale alcune delle odierne problematiche legate allo sviluppo di applicativi web possono essere risolte, inoltre sarà effettuata una comparazione con una seconda possibile soluzione.
Verranno definiti i concetti base del Test Driven Development, essendo quest'ultima la metodologia scelta per lo sviluppo del progetto di laurea.

1.1 TYPESCRIPT

TypeScript viene sviluppato principalmente da Microsoft sin dal 2012, il suo codice è rilasciato sotto licenza Apache 2.0 e disponibile su GitHub, il che ha permesso a molti sviluppatori ed aziende di contribuire al suo sviluppo.

TypeScript è basato su JavaScript, ed è un super-set di ECMASCRIPT2015, il che comporta che codice JavaScript sintatticamente corretto rispetto alle specifiche del linguaggio sia anche codice TypeScript valido.
Rispetto al JavaScript, aggiunge molteplici funzionalità che saranno analizzate nel corso di questo capitolo, la più importante è sicuramente l'aggiunta di tipi statici.
È fondamentale sottolineare che dopo il processo di traspilazione da codice TypeScript a codice JavaScript non vi saranno tracce dei tipi utilizzati, inoltre non esisteranno rappresentazioni dei tipi in fase di esecuzione e quindi nessun controllo di tipo in fase di esecuzione potrà essere effettuato.

1.1.1 *Tipi*

Nel linguaggio TypeScript è più corretto pensare ad un tipo come ad un insieme di valori che condividono qualcosa in comune, ad esempio un valore che può essere sia una stringa che un intero appartiene ad un insieme dato dall'unione di *string* e *int*.

Type System

Nei type system nominali due tipi sono considerati uguali se hanno lo stesso nome, e un tipo T1 è considerato un sottotipo (immediato) di un tipo T2 se T1 è dichiarato esplicitamente come un sottotipo di T2, un esempio di linguaggio di programmazione che utilizza questo tipo di type system è Java.

Listing 1.1: Type system nominale in Java

```
class Person {
   public name: string;
}

class Employee extends Person {
   public salary: number;
}

class Manager extends Employee { }

class Product {
   public name: string;
   public price: number;
}
```

Nel codice precedente *Manager* è sia sottotipo di *Employee* che di *Person*, *Product* invece pur disponendo degli stessi membri non ha un legame di relazione con le altre classi.

TypeScript adotta un type system strutturale, questo significa che le relazioni fra i tipi sono determinate dalle proprietà che contengono e non dal fatto che siano state dichiarate con una particolare relazione.
Ciò implica quindi che x è compatibile con y solamente se y ha almeno gli stessi membri di x.

L'utilizzo di un type system strutturale permette a TypeScript una maggiore flessibilità rinunciando alla manutenibilità fornita proprio dal type system nominale, che potrebbe già in fase di compilazione generare un errore nel caso una relazione fra due oggetti non fosse rispettata [1].

Listing 1.2: Type system strutturale in TypeScript

```
interface Cat {
  name: string;
}
let curry = { name: "curry", owner: "Alessandra" };

let awesomeCat : Cat = curry
```

In questo caso il compilatore TypeScript controlla se ogni proprietà presente nel tipo *Cat* è presente anche nell'oggetto *curry*, questo controllo risulta positivo e l'assegnamento non produce alcun tipo di errore.

Strongly typed

TypeScript, a differenza di JavaScript, è un linguaggio *strongly typed* [2], questo permette regole di assegnazione fra tipi più rigide, che provocano errori in fase di compilazione nel caso in cui non siano rispettate.

Listing 1.3: Esempio di assegnazione di tipi

```
interface Cat {
  name: string;
}

let awesomeCat : Cat

let name: string = "Curry"

const MAX_VALUE : number = 4

let isThesis: boolean = "true" // Error: Type 'string' is not
    assignable to type 'boolean'.
```

JavaScript può essere considerato come un linguaggio *weakly typed*, nell'esecuzione di codice JavaScript vi possono essere risultati imprevedibili, addirittura errati o possono essere eseguite conversioni di tipo implicite

in fase di esecuzione.

Non poter definire tipi nel codice JavaScript porta con se molteplici problematiche, una di esse è l'impossibilità di conoscere il corretto funzionamento dell'applicativo se non in fase di esecuzione.

Listing 1.4: Esempio di assegnazione di tipi

```
let awesomeCat

let name = "Curry"

const MAX_VALUE = 4

name = MAX_VALUE // No Error
```

1.1.2 *Unione ed intersezione di tipi*

TypeScript permette la definizione di nuovi tipi, attraverso operazioni di unione e/o intersezione a partire da tipi precedentemente definiti.

Unione di tipi

La creazione di un nuovo tipo come unione di tipi già esistenti avviene mediante l'utilizzo dell'operatore `|`. Una variabile dichiarata utilizzando quest'ultimo tipo può assumere un valore congruo con uno dei tipi utilizzati nell'operazione di unione.

Listing 1.5: Dichiarazione di una variabile attraverso l'unione di molteplici tipi

```
let myUnionVar: string | number | boolean;

myUnionVar = "Giulio";

myUnionVar = 3;

myUnionVar = {}; // Type '{}' is not assignable to type 'string |
    number | boolean'.
```

TypeScript permette l'accesso solamente ai campi condivisi da tutti i tipi che formano l'unione.

Listing 1.6: Unione di tipi con campi condivisi

```
interface Bird {
  fly(): void;
  eat(): void;
}

interface Fish {
  swim(): void;
  eat(): void;
}

declare function getSmallPet(): Fish | Bird;

let pet : Fish | Bird = getSmallPet();
pet.eat() // il metodo eat risulta condiviso fra i due tipi, nessun
    errore
pet.fly(); // Property 'fly' does not exist on type 'Bird | Fish'
```

Per poter accedere a membri specifici di ogni tipo è necessario effettuare a priori un controllo del tipo.

Intersezione di tipi

L'intersezione di molteplici tipi esistenti, attraverso l'operatore *&*, permette la creazione di un nuovo tipo, in esso saranno presenti tutti i membri dei tipi coinvolti nell'intersezione.

Listing 1.7: Intersezione di tipi

```
interface ErrorHandling {
  code: number;
  error?: { message: string };
}

interface NewsData {
  news: {
   name: string,
   content: string,
   author: string
  }[]
}

type NewsResponse = NewsData & ErrorHandling;
```

1.1.3 *Proprietà opzionali*

Alcune proprietà all'interno di un'interfaccia possono essere opzionali, esse vengono definite posponendo *?* al nome della proprietà. TypeScript è in grado di suggerire una correzione relativa al nome della proprietà opzionale nel caso in cui fosse stato fatto un errore tipografico.

Listing 1.8: Proprietà opzionali di un'interfaccia

```
interface Point {
  x: number;
  y: number;
  name?: string;
}
```

1.1.4 *Interfacce per tipi indicizzabili*

Le interfacce possono essere utilizzate per descrivere valori che possono essere indicizzati.
I tipi indicizzabili possiedono una "index signature" che descrive il tipo del valore utilizzato per l'indicizzazione e ne specificano anche il tipo per gli oggetti che verranno restituiti durante l'operazione di indicizzazione. TypeScript supporta due tipi come "index signature", *number* e *string*, nel caso si utilizzi un valore di tipo *number* come indice, il valore restituito deve essere obbligatoriamente un sottotipo di quello che verrebbe restituito se fosse stato usato invece un indice di tipo *string*, questa limitazione deriva dal comportamento di JavaScript, esso infatti durante l'operazione di indicizzazione convertirà il numero in stringa.

Listing 1.9: Interfacce per descrivere tipi indicizzabili

```
interface Numbers {
  [index: string]: number;
  length: number;
}
```

1.1.5 *Tipi Generici*

TypeScript permette la creazione di un componente generico facendo ricorso al tipo *any*, ma in questo modo verranno perse tutte le informazioni

aggiuntive di cui si potrebbe disporre nel caso in cui si conoscesse il reale tipo dell'argomento o della funzione.
TypeScript supporta i generics, e la definizione del tipo di quest'ultimi può avvenire utilizzando <> oppure attraverso il *type argument inference* in cui il valore di ritorno, in partenza generico, viene determinato in base al tipo dell'argomento.

Listing 1.10: Dichiarazione di tipi generici

```
function identity<T>(arg: T): T {
  return arg;
}

let first = identity<string>("TypeScript");
let second = identity("TypeScript");

class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}
```

In alcune situazioni, seppur facendo uso dei tipi generici, può essere necessario accedere ad una proprietà sicuramente esistente per i tipi che essi rappresenteranno a runtime, il compilatore non ha informazioni sui tipi che saranno successivamente rappresentati e non permette di effettuare questo tipo di operazioni.
In questo caso TypeScript permette la definizione di alcuni vincoli in modo tale che i parametri accettati posseggano almeno la proprietà richiesta.

Listing 1.11: Vincoli per tipi generici

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
```

1.2 JSX

JSX è un'estensione della sintassi ECMAScript, simile a XML, senza alcuna semantica definita e viene processato da appositi pre-processori che trasformano i suoi token in codice ECMAScript standard [3].

Listing 1.12: Esempio di utilizzo di JSX

```
const helloDiv : HTMLDivElement =
  <div id="1">
    <span>
      Hello
    </span>
    World!
  </div>
```

TypeScript supporta l'incorporamento, il controllo del tipo e la compilazione di JSX. L'utilizzo di questa sintassi è limitata ai file con estensione ".tsx", se TypeScript è utilizzato al loro interno, altrimenti con estensione ".jsx" ed è richiesto che nel file tsconfig.json la proprietà *jsx* abbia valore *true*.

TypeScript ha 3 differenti modi per gestire JSX: *preserve*, *react* e *react-native*, esse sono sfruttate solamente in fase di compilazione e non di type checking.
La prima modalità, *preserve*, non trasforma il codice JSX poiché dovrà essere processato successivamente da altri tool come Babel, con la seconda e terza modalità il codice JSX viene trasformato in chiamate alla funzione *React.createElement()*.

Affinché si possa sfruttare il type checking, è doveroso fare una distinzione fra Intrinsic element e Value-based element.

1.2.1 *Intrinsic Elements*

Questo tipo di elementi, il cui nome inizia sempre con una lettera minuscola, vengono convertiti in stringhe quando usati attraverso la funzione *createElement* essi corrispondono a tag HTML elementari nell'ambito del DOM ed i loro parametri devono rispettare le specifiche.
Per effettuare l'operazione di type checking TypeScript utilizza l'interfaccia *IntrinsicElement* all'interno del namespace JSX, essa viene implemen-

tata da tutti i framework client-side che utilizzano JSX.

1.2.2 *Value-based Elements*

Questi elementi possono essere definiti attraverso funzioni, in questo caso si parla di "Function Component" oppure attraverso classi "Class Component", la convenzione vuole che il loro nome inizi sempre con la lettera maiuscola.
Per TypeScript sono inizialmente indistinguibili quando usati all'interno di una espressione JSX, dunque TypeScript prova inizialmente a risolverli come "Function Component" facendo uso degli overloads, in alternativa come "Class Component".

1.2.3 *Children Type Checking*

Dalla versione 2.3 di TypeScript, è possibile effettuare un controllo di tipo sulla proprietà *children*, la sua struttura è definita all'interno dell'interfaccia *JSX.ElementChildrenAttribute*, questa proprietà sarà inclusa all'interno degli attributi del componente in modo totalmente automatico.

1.3 DOM

Il DOM, Document Object Model, è un API che definisce la struttura logica ed il modo in cui è possibile accedere e manipolare un documento HTML o XML [4].
Attraverso questa API possono essere creati documenti, navigare la struttura di essi, effettuare operazioni di aggiunta, modifica e rimozione di elementi.
Inoltre la sue specifiche sono definite attraverso il consorzio Object Management Group e godono di indipendenza dal linguaggio di programmazione in cui le API vengono consumate.

Nelle moderne applicazioni web alcune operazioni di interazione con il DOM possono risultare particolarmente onerose, causare rallentamenti o utilizzi anomali della CPU, alcuni esempi sono la creazione di un intero albero di elementi, un elevato numero di Repaint e Reflow causati dall'esecuzione di uno script o la ricerca di specifici nodi all'interno del documento.
Nel corso di questa sezione verranno introdotte alcune di queste proble-

matiche, in modo da evidenziare quali sono le possibili soluzioni e quale è stata adottata nella tesi di laurea.

1.3.1 *Repaint*

L'operazione di Repaint viene innescata nel momento in cui un elemento all'interno del documento cambia il proprio stato di visibilità, da visibile ad invisibile e viceversa, senza però alterare il layout della pagina [5].
Alcuni esempi di operazioni che innescano il Repaint sono l'aggiunta di outline ad un elemento o la modifica del background.

Questa operazione risulta particolarmente onerosa poiché l'engine del browser dovrà effettuare una ricerca fra tutti gli elementi del documento per determinare quali elementi cambieranno il proprio stato di visibilità e definire quello che sarà l'aspetto della pagina web dopo la modifica.

1.3.2 *Reflow*

Con il termine Reflow si intende l'operazione con la quale l'engine del browser ricalcola la posizione e le geometrie degli elementi per effettuare il rerendering di alcuni elementi o di tutto il documento [6].
Questa operazione viene innescata nei seguenti casi: un elemento del documento viene manipolato, un nuovo elemento viene aggiunto, lo stile di un elemento varia e questa modifica apporta un cambiamento al layout, quando viene apportato un cambiamento alla proprietà *className* di un elemento ed infine quando la dimensione della finestra del browser cambia [5].
L'operazione di Reflow su uno specifico elemento comporta la stessa operazione anche per gli elementi successivi, per i suoi elementi figli e anche per il padre, questo è necessario al fine di garantire la correttezza del layout dopo la modifica dell'elemento, infine viene eseguita una operazione di repaint.

Listing 1.13: Struttura HTML ed Operazione di Reflow

```
<body>
  <div>
    <div> <!-- Parent -->
      <p>Hello World!</p>
      <p>Aloha honua</p>
      <p>ao ora</p>
```

```
    </div>
    <h2>TDD</h2>
    <h3>DOM</h3>
  </div>
</body>
```

Nel codice HTML sopra, la modifica della proprietà *className* dell'elemento Parent provocherà un Reflow degli elementi h2 ed h3 poiché successivi, di tutti gli elementi figli paragraph e del padre.

Il Reflow è un'operazione fondamentale per la corretta visualizzazione della pagina web e non può essere evitata, si possono però effettuare manipolazioni ad elementi del DOM seguendo alcuni criteri che possono ridurre il numero di Reflow richiesti.
Alcuni esempi di ottimizzazioni possono essere:

- Effettuare una copia dell'elemento che deve essere manipolato ed effettuare tutte le manipolazioni sulla copia, infine verrà effettuato uno swap fra l'elemento presente nel DOM ed il suo clone.
- La memorizzazione in una variaible di un elemento del DOM al fine di evitare successive ricerche per lo stesso elemento.
- La memorizzazione di valori come larghezza ed altezza di un elemento in variabili, ogni accesso al proprietà come *offsetWidth*, *marginLeft*, *left* causano reflow.

Nelle moderne applicazioni web, altamente interattive, gli elementi del DOM cambiano per rispecchiare lo stato dell'applicazione; i frequenti cambiamenti potrebbero portare ad un peggioramento delle performance e la gestione di essi diverrebbe troppo complessa utilizzando solamente le API messe a disposizione dal DOM, inoltre la manutenibilità e la scalabilità dell'applicazione ne risentirebbero.

Nella tabella 1 viene mostrato come le prestazioni delle operazioni di Reflow cambiano in base al browser utilizzato ed alla proprietà che viene manipolata, le misurazioni sono state effettuate su 1000 regole e 1x è un arco temporale che varia da 1 a 6 secondi.

Tabella 1: Prestazioni delle operazioni di Reflow su differenti browser

Reflow			
Proprietà	Chrome v2	Firefox v3	IE 8
className	1x	1x	1x
display none	-	-	-
display default	1x	2x	1x
visibility hidden	1x	1x	1x
visibility visible	1x	1x	1x
padding	-	1x	4x
font size	1x	2x	1x

1.4 VIRTUAL DOM

Il concetto di virtual DOM è divenuto molto popolare ed utilizzato nell'ambito web grazie a ReactJs, una libreria open source per la realizzazione di interfacce grafiche realizzata e sviluppata da Facebook dal 2013 ed arrivata oggi alla versione 17, questa libreria viene utilizzata da aziende come Amazon, Instagram, Netflix, Uber e molte altre.

La sua ideazione deriva dalla necessità, da parte di Facebook, di mostrare pubblicità agli utenti e sincronizzare la UI con lo stato dell'applicazione. Il suo concetto si basa sulla rappresentazione in memoria, per esempio utilizzando un oggetto javascript, del DOM senza costruirne una reale istanza, utilizzando l'albero come struttura dati.

Listing 1.14: Implementazione semplificata di un elemento rappresentato attraverso il VDOM

```
let vnode : VNode = {
      type: 'div' ,
      props: {
        id: 1
        children: [ {
          type: 'h1',
          props:{
            className: "title",
            children: []
          }
```

```
        }
      }
    }
```

Il codice 1.14 mostra la rappresentazione dei seguenti elementi HTML attraverso una versione semplificata del Virtual DOM.

Listing 1.15: Struttura HTML reale precedentemente rappresentata attraverso il VDOM

```
<div id='1'>
  <h1 class='title'></h1>
</div>
```

Ogni elemento del Virtual DOM prende il nome di componente, essi vengono combinati per formare un albero di elementi che sarà una fedele rappresentazione del DOM.

La sua rappresentazione in memoria rende possibili molteplici modifiche senza causare operazioni di Repaint o Reflow per ognuna di esse, inoltre è possibile raggruppare un insieme di modifiche al fine di effettuare un'unica operazione di manipolazione del DOM, questa operazione prende il nome di *batching*, essa evita un peggioramento delle performance della pagina web garantendo una buona esperienza utente.

Un nuova versione del Virtual DOM viene creata ogni qualvolta debba essere effettuato un cambiamento nel DOM, in seguito attraverso un algoritmo di *diffing* le due versioni del Virtual DOM vengono comparate al fine di trovare le minime modifiche necessarie da eseguire sul DOM.

Listing 1.16: Rappresentazione dei due stati del Virtual DOM

```
let oldVNode : VNode = {
       type: 'div' ,
       props: {
         id: 1
         children: [ {
           type: 'h1',
           props:{
             className: 'title',
             children: []
           }
         }
       }
```

```
    }

let nextVNode : VNode = {
      type: 'span' ,
      props: {
        className: 'container',
        children: [ {
          type: 'h3',
          props:{
            id:1,
            children: []
          }
        },
        {
          type: 'p',
          props:{
            children: []
          }
        }
      }
    }
```

Nel corso di questo progetto di laurea verrà implementata una versione semplificata del Virtual DOM, al quale saranno aggiunte specifiche proprietà derivate dalle scelte progettuali effettuate, che verranno discusse nei prossimi capitoli.

1.5 ALTERNATIVE AL VIRTUAL DOM

Una delle più conosciute alternative al concetto del Virtual DOM è il Dirty Checking utilizzato da AngularJS, un framework open source per lo sviluppo di applicazioni web realizzato da Google nel 2010.
Questa tecnica consiste nell'eseguire un ciclo, chiamato *digest cycle*, che si occupa di controllare se alcuni dati dell'applicazione hanno subito variazioni, questo è possibile poiché i dati che devono essere osservati per eventuali modifiche sono preventivamente registrati attraverso specifiche istruzioni.
Il Digest Cycle ha un costo computazionale elevato, per ogni dato sotto osservazione deve essere effettuata una comparazione con la versione precedente di quest'ultimo. Esistono alcuni casi d'uso nei quali l'utilizzo di questo meccanismo offre migliori risultati in termini di performance

rispetto all'utilizzo del Virtual Dom, in altri invece la sua mancanza produce risultati molto deludenti.

Nella seguente tabella vengono riportati i tempi di esecuzione di alcune operazioni effettuate utilizzando ReactJS (v16.12.0) e Angular (v8.2.14), questi benchmark sono stati effettuati utilizzando il browser web Chrome. [7]

Tabella 2: Comparazione prestazionale tra ReactJS e Angular

Framework/ Operazioni	ReactJs	Angular
Manipolazione di un elemento in una collezione di 1000.	~16.58ms	~6.08ms
Modifica di ogni elemento p per ogni elemento div in una collezione di 1000.	~17.86ms	~896.76ms
Eliminazione di un elemento div con il figlio p da una collezione di 1000 elementi.	~16.54ms	~0.09ms
Eliminazione di ogni elemento div e del figlio p in una collezione di 1000 elementi.	~7.39ms	~23.83ms

Nel primo benchmark, nel quale si effettua la manipolazione di un singolo elemento *div* fra altri 1000 elementi, Angular utilizzando il Dirty Checking risulta essere circa 10ms più veloce rispetto ad ReactJS, questa elevata differenza fra i due risultati può essere associata ai passi intermedi eseguiti sul Virtual DOM prima di essere riportati nel DOM.
Nel secondo benchmark, nel quale ogni elemento *p* per ciascun elemento *div* viene modificato, si verifica un caso molto interessante, ReactJS riesce ad eseguire le operazioni richieste con una notevole disparità di tempo, questa notevole differenza ancora una volta può essere associata all'utilizzo del Virtual DOM, le operazioni vengono prima eseguite in memoria e solamente dopo effettuate sul DOM, utilizzando la tecnica del batching.
Nel terzo benchmark si nota ancora una volta come ReactJS debba modificare prima la rappresentazione in memoria del DOM prima di apportare effettivamente la modifica, nell'ultimo caso la manipolazione riguarda molteplici elementi e il Virtual DOM risulta più efficiente.
Dai risultati precedentemente elencati l'utilizzo del Virtual DOM risulta essere un buon compromesso in svariati casi d'uso, inoltre la sua facilità di comprensione ed implementazione lo rende una delle soluzioni più utilizzate negli ultimi anni.

1.6 TEST DRIVEN DEVELOPMENT

Il Test Driven Development, TDD, è una metodologia di sviluppo nel quale i requisiti software vengono convertiti prima in test case.
Utilizzare questa metodologia per lo sviluppo di un applicativo si converte in molteplici benefici, il codice risulterà più facilmente leggibile, modulare e debolmente accoppiato [8].
Lo sviluppo di un software utilizzando la metodologia TDD si basa su un ciclo di sviluppo formato da 3 differenti fasi.

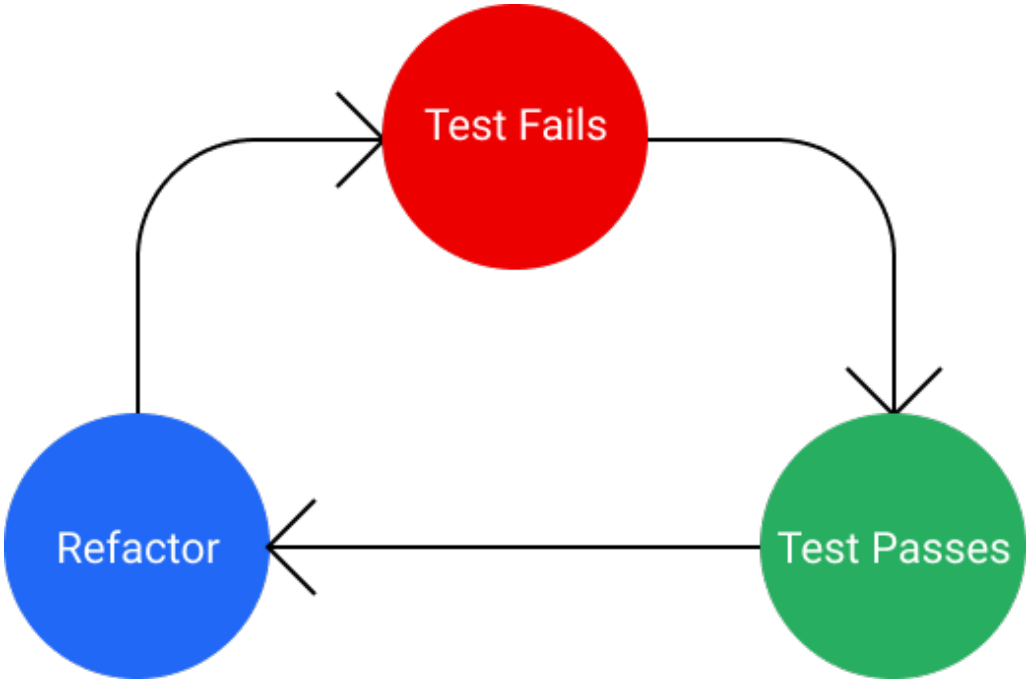

Figura 1: Ciclo del Test Driven Development composto da 3 fasi

Nella prima fase, detta Red phase, verrà scritto un test per una feature non ancora implementata, questo test fallirà e consentirà di passare alla seconda fase.
Nella seconda fase, detta Green phase, può essere scritto solo il codice strettamente necessario affinche il test precedentemente scritto abbia esito positivo.
La terza e ultima fase prende il nome di Refactor phase, a differenza delle prima due che risultano obbligatorie, questa può essere considerata

facoltativa; in essa è possibile effettuare un'operazione di refactoring del codice assicurandosi che i test precedentemente scritti continuino ad avere un esito positivo [9].

Il processo del Test Driven Development viene descritto da Robert C Martin [10], attraverso 3 leggi:

- Non è consentito scrivere codice di produzione a meno che non serva per superare lo unit test che aveva esito negativo.
- Non è consentito scrivere più di un Unit Test fallimentare per volta.
- Non è consentito scrivere codice aggiuntivo se non di quello strettamente necessario affinché lo unit test abbia esito positivo.

1.6.1 *Code coverage*

La code coverage misura quante linee di codice sono state eseguite durante l'esecuzione dell'applicativo, essa risulta quindi una metrica molto importante da consultare durante lo sviluppo di quest'ultimo. Se l'applicazione in esecuzione durante la misurazione della copertura del codice è una suite di test si parlerà di Test Coverage [9]; successivamente faremo riferimento alla Test Coverage attraverso la dicitura Code Coverage.
Un alto valore di questa misurazione significherà che molte parti del codice sono state eseguite durante i test, il che si riconduce alla possibilità di avere un minor numero di bug in fase di esecuzione dell'applicativo.
L'utilizzo della metodologia Test Driven Development permette il raggiungimento del 100% della Code Coverage, è importante notare che pur essendo una metrica fondamentale, da sola non garantisce che il codice non presenti bug, i test scritti potrebbero eseguire asserzioni sbagliate o non eseguire asserzioni.

1.6.2 *Mutation Testing*

La code coverage è una metrica importante, ma non può garantire che in fase di esecuzione dell'applicativo non si presentino errori.
Attraverso dei mutation testing framework il SUT (system under test) viene modificato, questa modifica prende il nome di mutante.
La suite di test viene eseguita nuovamente, nel caso almeno un test fallisca, per il SUT modificato, si dice che il mutante è stato ucciso altrimenti il mutante è sopravvissuto. In quest'ultimo caso i test non

coprono tutta la complessità del codice poiché non sono riusciti a rilevare il cambiamento, alcuni esempi di mutazioni sono i seguenti:

- L'operatore && viene sostituito con || e viceversa
- Una variabile booleana viene sostituita con true o false
- Completa rimozione di una chiamata a funzione

Durante lo svolgimento di questa tesi la scelta del mutation framework è ricaduta su StrykerJS l'unico framework attualmente disponibile e supportato per JavaScript e Typescript.

1.7 CONTINUOUS INTEGRATION

Nello sviluppo del software con il termine Continuous Integration si intende la pratica con la quale membri di un team allineano le loro modifiche frequentemente, almeno una volta al giorno [11].
Ogni integrazione viene verificata da una build automatica, in questo modo è possibile conoscere se l'applicativo presenta eventuali bug in fase di compilazione, inoltre la codebase viene completamente testata eseguendo tutti i test disponibili [9].

La Continuous Integration necessità di un meccanismo di compilazione automatica dell'applicativo, nel caso di progetti TypeScript i comandi necessari, ad effettuare la build e l'esecuzione dei test, vengono dichiarati all'interno del file *package.json* sotto la proprietà *scripts*.
Solitamente questo tipo di operazioni avvengono sul Continuous Integration Server al verificarsi di determinati eventi, come il push di un commit o l'apertura di una Pull Request, questa operazione permette di dare una certa confidenza sul fatto che l'applicativo continuerà a funzionare anche dopo la conclusione dell'operazione di merge.
La build ed i test eseguiti sul CI Server verranno conclusi in minor tempo rispetto all'esecuzione degli stessi in locale, nel caso di complesse e ampie codebase in cui queste operazioni potrebbero impiegare svariati minuti lo sviluppatore può continuare a lavorare venendo informato dell'esito delle operazioni al completamento di esse.

Durante questo capitolo è stata data una breve introduzione a TypeScript, un linguaggio Strongly Typed, ed alle principali funzionalità che verranno utilizzate durante la codifica del progetto di laurea, proprio

per quest'ultimo si è scelto di utilizzare la metodologia Test Driven Development, convertendo in primo luogo i requisiti software in test case ed utilizzando la Continuous Integration. Sono state descritte alcune delle odierne problematiche dello sviluppo di applicativi web ed alcune possibili soluzioni come il Virtual DOM e il Dirty Checking, il Virtual DOM sarà un concetto implementato nel progetto di laurea in modo semplificato per le scelte progettuali effettuate.

2

FRAMEWORK

Nel corso del corrente capitolo verranno descritti i moduli e le strutture dati che compongono il framework web realizzato, le funzionalità implementate e le scelte progettuali effettuate.
Si farà vasto uso di immagini per dare una rappresentazione visiva dell'algoritmo di diffing descrritto nell'apposita sezione.

2.1 JSX, CREATEELEMENT E VIRTUAL NODE

Come introdotto nel primo capitolo, JSX essendo un estensione del linguaggio JavaScript può essere utilizzato dopo l'esecuzione di un pre-processore che ne trasforma i token in codice ECMAScript standard.

Listing 2.1: Esempio di token JSX

```
const divElement: HTMLDivElement =
  <div id="1">
    <span></span>
  </div>
```

Il pre-processore trasforma un token JSX in una chiamata ad una *factory function*, quest'ultima restituisce una rappresentazione del token attraverso un dizionario in cui vengono specificati il tipo dell'elemento HTML o la definizione del Class Component attraverso la chiave *type*, gli attributi specificati ed i suoi figli vengono memorizzati rispettivamente utilizzando le chiavi *props* e *children*, questo dizionario rappresenta una istanza del tipo *VNode*.

Listing 2.2: Esempio di trasformazione di un token JSX attraverso la factory function

```
//Token JSX
const divElement: HTMLDivElement =
```

```
  <div id="1">
    <span></span>
  </div>

// Struttura della chiamata alla factory function
createElement(
  "div",
  {
    id: "1",
  },
  createElement("span", null)
);

// Valore ritornato dalla factory function rappresentate un VNode
{
  type: "div",
  props: {
    id: "1",
    children: [
      {
        type: "span",
        props: {
          children: []
        }
      }
    ]
  }
}
```

Come mostrato in 2.2, la funzione *createElement* restituisce un Virtual Node che rappresenterà l'elemento o un insieme di elementi HTML in memoria, l'insieme dei Virtual Node (VNode) e le loro relazioni forma il Virtual DOM.
La struttura di un Virtual Node non è predefinita e non esistono linee guida per la sua definizione, esso può essere strutturato in base alle esigenze e scelte progettuali, la sua definizione e implementazione non sarà accessibile né all'utente finale né allo sviluppatore che utilizzerà il framework, ma verrà utilizzato principalmente dall'algoritmo di diffing analizzato in seguito.
Nel framework sviluppato si è scelto di aggiungere altre tre proprietà al Virtual Node oltre alle già sopracitate *type props*, *children*.
La prima è *dom* la quale conterrà un riferimento all'istanza dell'elemento

presente nel DOM , la seconda è *component* che viene utilizzata solamente nei casi in cui venga rappresentato un Class Component e contiene un riferimento all'istanza di quest'ultimo.
L'ultima è *reusable* ed indica se un VNode è considerato riutilizzabile dall'algoritmo di diffing, e verrà maggiormente dettagliata nella sottosezione Differ del corrente capitolo.
Si è scelto di posizionare la proprietà *children* all'interno della proprietà *props*, questa scelta è stata attuata per ottenere una compatibilità, seppure limitata, con ReactJS pur non essendo questo un aspetto su cui il progetto di laurea si basa.

Per l'implementazione della *factory function* non è stata definita una classe, ma solamente la funzione *createElement* descritta precedentemente, questa scelta garantisce una maggiore compatibilità con molteplici pre-processori poiché risulta necessario specificarla in appositi file di configurazione o nei file dove JSX viene utilizzato.

2.2 RENDERER

Il modulo Renderer definisce l'interfaccia pubblica attraverso la quale viene eseguita la renderizzazione dell'applicazione web, l'operazione con la quale a partire dal codice JSX l'albero di elementi HTML viene inserito nel DOM.
Durante il primo render effettuato viene memorizzata la prima versione del Virtual DOM, attraverso essa sarà possibile, nei futuri re-render, effettuare una comparazione per variare solamente i Virtual Node che hanno subito una modifica, aggiungerne di nuovi o rimuoverne di esistenti. Il metodo *render* fornito dal framework web renderizza l'albero di elementi HTML all'interno di un elemento HTML che prende il nome di *root*, al suo interno l'intera gestione degli elementi è delegata al framework e non possono essere effettuate manipolazioni da parte dello sviluppatore.

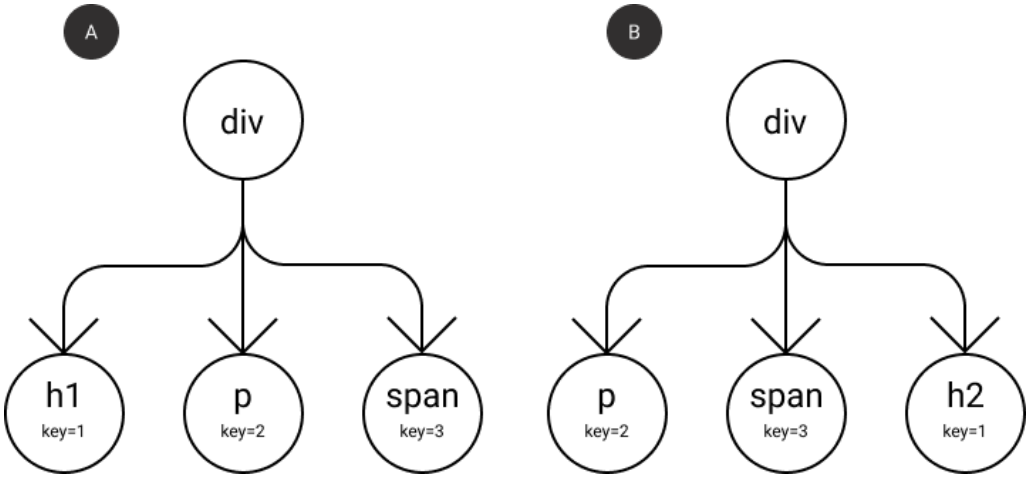

Figura 2: Rappresentazione dell'albero per l'attuale versione del Virtual DOM nel lato sinistro, e la versione successiva del Virtual DOM, causata dal re-render, in cui ogni elemento possiede l'attributo *key* nel lato destro

2.3 DIFFER

Il modulo Differ svolge un ruolo fondamentale all'interno del framework web, attraverso quest'ultimo, quando si verifica un'operazione di re-rendering, l'attuale versione del Virtual DOM, proveniente dal render precedente, e la successiva versione, proveniente dall'attuale render, vengono comparate al fine di identificare le minime variazioni da riflettere sul DOM, la comparazione avviene per ogni Virtual Node che compone il Virtual DOM.

La ricerca delle minime operazioni per la trasformazione di un albero, che è la struttura dati alla base di un Virtual DOM, in un'altro può avere una complessità nell'ordine di $O(n^3)$ con n il numero di elementi di cui l'albero è composto [12] [13].
Nelle moderne applicazioni web questo numero è molto elevato, è stata effettuata una query sui report sviluppati da HTTPArchive [14] per l'anno 2020, un progetto open source che effettua un operazione di crawling su un vasto insieme di siti web periodicamente, e la media dei nodi presenti nel DOM è circa 880.6788.

Per poter effettuare l'operazione di trasformazione in modo più efficiente si è fatto uso delle stesse assunzioni utilizzate da ReactJS:

- Due elementi di tipo diverso produrranno alberi diversi.

- È possibile attraverso l'utilizzo della proprietà *key* suggerire all'algoritmo di diffing quali elementi potrebbero mantenersi stabili tra le diverse renderizzazioni.

Esse definiscono le specifiche dell'algoritmo per la ricerca del miglior nodo valido, allo stesso livello nei i due alberi, con cui effettuare il confronto per rilevare eventuali modifiche.
Definiamo valido, per il riutilizzo, un Virtual Node che eviti la rimozione di un elemento o di un albero di elementi presenti nel DOM e che quindi possa essere riutilizzato.

Quando viene effettuata un'operazione di re-rendering, ogni nodo della nuova versione del Virtual DOM viene comparato con tutti i nodi presenti, nello stesso livello, nell'attuale versione di quest'ultimo.
Nel caso i VNode non contengano la proprietà *key*, per determinare se un nodo è valido si effettua una comparazione del campo *type* all'interno di essi, se esso coincide il nodo viene selezionato e la ricerca viene interrotta, altrimenti si procede con il confronto del nodo successivo.

Nel caso in cui entrambi i Virtual Node presentino la proprietà *key* è richiesto che esse assumano ugual valore e che il il valore delle chiavi *type* non differisca, se le due richieste sono soddisfatte il nodo viene considerato valido, la proprietà *reusable* viene impostata a *true* e sarà riutilizzato.

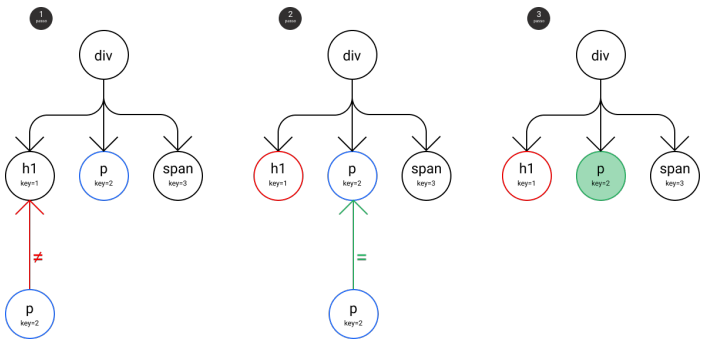

Figura 3: Operazione di confronto, con Virtual Node riutilizzabile in seconda posizione

Nel caso in cui il confronto abbia esisto negativo, la proprietà *reusable*, del VNode considerato non riutilizzabile, viene impostato a *false*,

l'istanza dell'elemento HTML associato non viene immediatamente rimosso dal DOM, poiché il VNode potrebbe risultare riutilizzabile per uno dei prossimi nodi, presenti allo stesso livello, nella nuova versione del Virtual DOM.
Quando la serie di controlli si conclude, l'elemento HTML associato ad ogni Virtual Node con la proprietà *reusable* con valore *false* viene rimosso dal DOM.

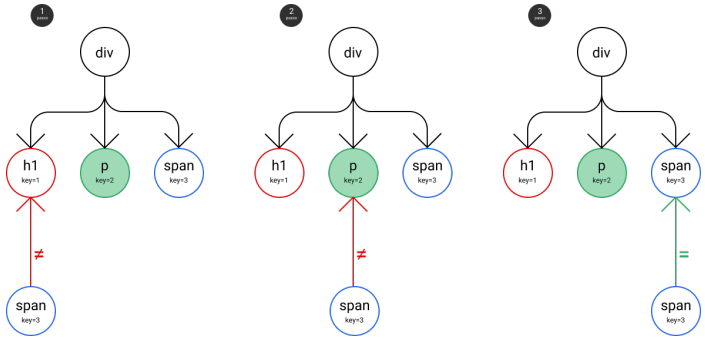

Figura 4: Operazione di confronto, con Virtual Node riutilizzabile in terza posizione

Nel caso in cui nessun Virtual Node possa essere considerato valido, ne verrà creato uno nuovo con tutte le proprietà vuote, si sta trattando un elemento non precedentemente contenuto nel DOM, questo elemento verrà istanziato ed aggiunto all'interno del DOM.

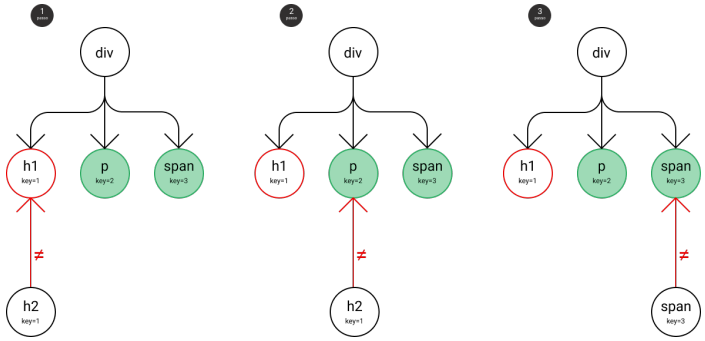

Figura 5: Operazione di confronto, nessun Virtual Node riutilizzabile, l'istanza dell'elemento HTML associato al VNode per h1 verrà rimosso dal DOM

La possibilità di assegnazione di un attributo *key* ad elementi, solitamente situati in una lista, permette di mitigare le problematiche elencate nel capitolo sullo stato dell'arte 1.3.
L'algoritmo di diffing è in grado di riconoscere il caso in cui gli elementi abbiano subito solamente una variazione della loro posizione nel DOM, evitando operazioni come la loro rimozione e il successivo re-inserimento, considerate molto onerose.

Quando sono gli attributi a differire fra le due versioni dei Virtual Node, come *id className customProp*, all'elemento del DOM associato, vengono applicati i nuovi attributi e rimossi tutti gli attributi che non sono presenti nella nuova versione del Virtual Node.

2.4 CLASS COMPONENT

L'utilizzo di framework web permette, allo sviluppatore che ne fa uso, di creare dei Componenti Custom, essi rappresentano un elemento o un insieme di elementi HTML, incapsulando logica e funzionalità che ne determinano il comportamento ed il contenuto, essi rientrano nei Value-based elements introdotti nella sezione JSX del primo capitolo.

Se definiti ed utilizzati correttamente permettono di rispettare i principi SOLID [15], effettuare più facilmente ed efficacemente dei test sul corretto funzionamento dell'applicativo web e una maggiore riutilizzabilità del codice.
Nel corrente progetto di laurea si è deciso di permettere la creazione di Value-based elements attraverso l'estensione di una classe, definita attraverso il nome AleliComponent, fornita dal framework stesso.
Essa definisce alcuni attributi e metodi standard, come lo stato che viene definito attraverso un dizionario, e il ciclo di vita del componente.

Con ciclo di vita si intende un insieme di metodi, ognuno dei quali è invocato in uno specifico momento o al verificarsi di un evento specifico, nel nostro caso esso è formato da 3 diversi metodi più il costruttore del Class Component

- *mounting*
- *render*
- *destroying*

Nel framework web realizzato, il ciclo di vita del componente è innescato dall'algoritmo di diffing.

Nel caso in cui un componente custom debba essere renderizzato per la sua prima volta, una sua istanza viene creata successivamente viene richiamato il metodo *mounting* , successivamente verrà invocato il metodo *render* che restituisce il Virtual Node rappresentante l'insieme di elementi HTML.

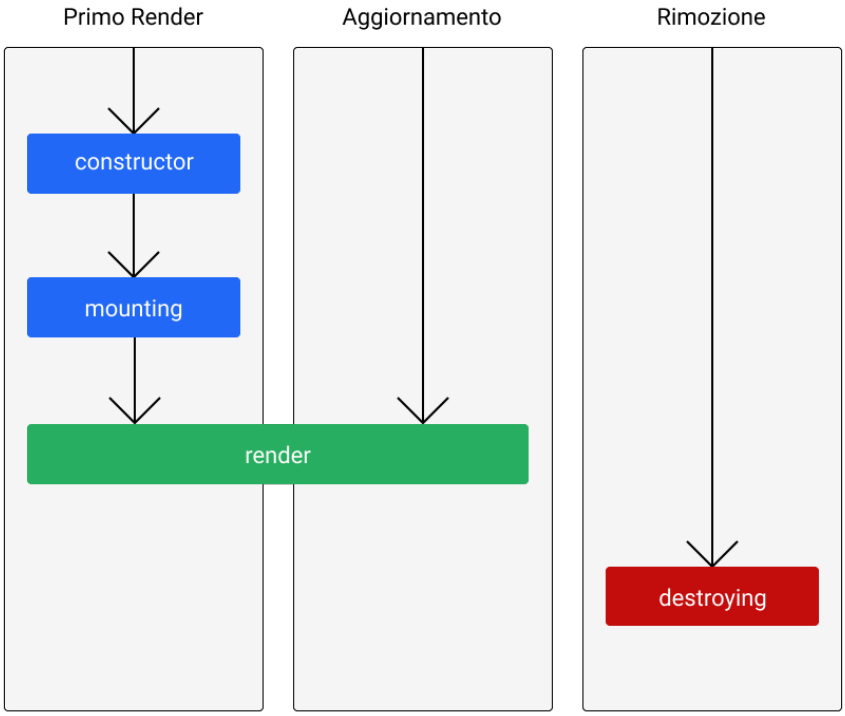

Figura 6: Ciclo di vita del componente

Solitamente lo scopo principale del costruttore di un Class Component consiste nell'inizializzare lo stato di quest'ultimo, attraverso il metodo *mounting* possono essere effettuate chiamate a specifiche API al fine di caricare dati che potrebbero essere necessari o qualsiasi operazione necessaria per il corretto funzionamento del componente
Infine attraverso il metodo *destroying* è possibile rimuovere risorse allocate, event listener precedentemente inizializzati o cancellare richieste di

rete ancora in corso.

Nel caso in cui il componente custom sia considerato non più valido dall'algoritmo di diffing, è necessaria l'invocazione del metodo *destroying* e la rimozione di tutto l'albero di elementi presenti nel DOM, quest'ultima operazione viene effettuata rimuovendo l'istanza dell'elemento HTML associato al VNode rappresentante il custom component.

Nel capitolo appena concluso si è data una panoramica generale dei moduli e delle funzionalità implementate, si è descritto l'algoritmo di diffing, considerato il core del framework web poiché, attraverso le comparazioni effettuate, mantiene sincronizzato il Virtual DOM con la UI della pagina web.
Attraverso una maggiore ottimizzazione del modulo Differ è possibile ottenere reali benefici sulle prestazioni dell'intero framework web, inoltre è possibile, e consigliabile, modificare la struttura ed implementazione del Virtual Node proprio per ottimizzare specifiche operazioni dell'algoritmo di diffing.

3

ANALISI DEL CODICE

Nel seguente capitolo verranno analizzate le porzioni di codice ritenute più importanti al fine di descrivere in dettaglio il funzionamento del framework web realizzato, fra queste vi sono tipi, interfacce e le classi dichiarate.
Inoltre verrà descritta la motivazione dei design pattern utilizzati e alcuni metodi di JavaScript di particolare importanza.

3.1 TIPI

3.1.1 *Virtual Node*

Per definire il tipo VNode, che verrà ampiamente utilizzato nel resto del framework, si è scelto di utilizzare un'interfaccia in modo che sia estendibile in futuro o se fosse necessario in seguito utilizzare il *declaration merging* [16], queste operazioni non sarebbero permesse se avessimo definito il tipo attraverso i Type Aliases di Typescript.

Listing 3.1: Dichiarazione del tipo VNode utilizzando un'interfaccia

```
interface VNode<T = {}> {
  type: string | { new(): Component };
  props: T & { children: Children, [other: string]: any }
  dom?: HTMLElement | Text
  component?: Component
  reusable?: Boolean | null;
}
```

Alcune proprietà vengono dichiarate come opzionali, esse assumono valore solo in determinate circostanze nel ciclo di vita dell'applicativo.

Per molti valori assunti, da speficiche proprietà, si è fatto uso dell'o-

peratore unione, questo permette ad esse di poter assumere valori di differente tipo senza incorrere in errori in fase di compilazione ed un miglior suggerimento attraverso l'IDE utilizzato.
Vi sono alcune particolarità nelle proprietà *type* e *props*, nel primo caso si usa *new(): Component* come tipo per indicare la definizione del Class Component, nel secondo caso viene fatto uso dei generics perché tutte le proprietà, esclusa *children*, vengono definite su un componente dallo sviluppatore, di esse non si conosce né la chiave, né il suo valore e neppure il tipo.
Inoltre si è fatto uso delle *index signature* per poter accedere al valore di una proprietà attraverso la sua chiave, operazione altrimenti non permessa da Typescript.

3.1.2 *Custom HTML Element*

Il tipo CustomHTMLElement è stato definito come estensione del preesistente tipo HTMLElement presente in Typescript, ad esso viene aggiunta la proprietà opzionale *_vnode*, utilizzata per la memorizzazione dell'intero Virtual DOM e si fa utilizzo delle index signature.
Anteponendo il carattere _ al nome della proprietà si segnala che essa dovrà essere trattata come privata e che non dovrebbero essere effettuate modifiche al suo valore, essa però rimane visibile ed accessibile a run time attraverso l'istanza dell'elemento HTML. Di particolare importanza in questo caso sono le index signature, permettendo di aggiungere proprietà custom, definite dallo sviluppatore e situate nel dizionario props all'interno del VNode, all'istanza di un elemento HTML, TypeScript permetterebbe un assegnazione di valore solamente alle proprietà definite nell'interfaccia HTMLElement.

Listing 3.2: Dichiarazione del tipo CustomHTMLElement utilizzando un'interfaccia

```
interface CustomHTMLElement extends HTMLElement{
  [other: string]: any
  _vnode?: VNode
}
```

3.1.3 *Class State*

La definizione dell'interfaccia ClassState permette di accedere al valore di una chiave presente nello stato del Custom Component poiché come segnalato precedentemente TypeScript non avrebbe permesso di accedere al valore di una chiave.

Listing 3.3: Dichiarazione del tipo ClassState utilizzando un'interfaccia

```
default interface ClassState {
  [key: string]: any;
}
```

3.2 FUNZIONI, INTERFACCE E CLASSI

3.2.1 *createElement*

Come descritto nella sezione 1.2, i token JSX vengono pre-processati dalla funzione *createElement*, quest'ultima prende in ingresso tre argomenti, il tipo dell'elemento, l'insieme degli attributi e proprietà del componente e infine i figli, che saranno a loro volta invocazioni alla createElement.

Listing 3.4: Implementazione della funzione createElement

```
function createElement<T = {}>(
  type: VNode["type"],
  props: VNode["props"] & T,
  ...jsxChildren: Children[]
): VNode<T> {
  const flattenChildren : Array<Children> =
      Array.prototype.concat.apply([], jsxChildren)

  const children : Array<Children> =
   flattenChildren.map((child) => {
      return !!child && typeof child === "object"
          ? child
          : {
            type: "$TEXT",
            props: { textValue: child, children: [] }
          };
  });
```

```
  return {
    type: type,
    props: { ...props, children },
  };
}
```

Inizialmente viene eseguita una normalizzazione dell'array *jsxChildren*. Attraverso l'utilizzo dei metodi *apply* [17] e *concat* [18] viene creato un array, *flattenChildren*, contenente tutti elementi concatenati in modo ricorsivo fino al primo livello di profondità.

Il metodo *apply* invoca *concat* fornendo come parametri un valore rappresentante il *this*, in questo caso un array vuoto, e l'array da concatenare reso monodimensionale. Sull'array *flattenChildren* viene invocato il metodo *map*, esso itera quest'ultimo array ed usa ogni elemento iterato come parametro della funzione passatagli come argomento, i risultati di quest'ultima formano l'array normalizzato.

In questo caso la funzione, passata come argomento al metodo map, restituisce l'elemento, ricevuto come parametro, se risulta essere un dizionario, altrimenti crea un nuovo dizionario specificando *type* con il valore di *$TEXT* e *textValue* con valore l'elemento ricevuto come parametro.
Nell'ultimo caso descritto, viene impostato a *type* il valore *$TEXT* per stabilire che il Virtual Node rappresenterà un nodo di testo, questa differenza riguarderà come verrà instanziato l'elemento HTML e come verrà gestita l'aggiunta o la rimozione dei suoi attributi.
Infine viene restituito un dizionario VNode, in cui sono specificati il tipo, le proprietà ed i figli normalizzati.

3.2.2 *Component e AleliComponent*

Attraverso l'interfaccia *Component* vengono definite le firme dei metodi che formano l'API pubbliche di un Custom Component.

Listing 3.5: Dichiarazione dell'interfaccia Component

```
interface Component {
  render(props: VNode["props"]) : VNode
  mounting() : void
  isMounted() : boolean
  mount() : void
  destroying() : void
```

```
  isDestroyed(): boolean
  destroy() : void
  getState(): ClassState
  getValueFromState(key: string) : any
  setState(newState: ClassState): void
}
```

La classe astratta *AleliComponent* implementa quest'ultima interfaccia, e fornisce un'implementazione standard per alcuni dei suoi metodi, per altri viene usato il modificatore abstract poiché devono obbligatoriamente essere implementati dallo sviluppatore che definisce un Custom Component.

Listing 3.6: Dichiarazione della classe AleliComponent implementando l'interfaccia Component

```
abstract class AleliComponent implements Component {
  protected readonly state: ClassState;
  private mounted: boolean;
  private destroyed: boolean;

  constructor() {
    this.mounted = false
    this.destroyed = false
    this.state = {};
  }

  // Alcuni metodi sono stati rimossi per abbreviare la porzione di
     codice

  abstract destroying(): void

  abstract mounting(): void

  abstract render(props: VNode["props"]): VNode<{}>;
}
```

Per quanto riguardo lo stato del componente, si è scelto di usare il modificatore *readonly*, non sarà quindi permesso nessun assegnamento al membro *state* al di fuori del costruttore, nel quale viene inizializzato. Negare possibili assegnamenti di valori è importante perché tutte le operazioni di aggiornamento dello stato devono essere obbligatoriamente

effettuate attraverso il metodo *setState*, in futuro potrebbe essere possibile implementando nuove funzionalità, renderizzare nuovamente il componente quando si verifica un aggiornamento dello stato.

Un metodo particolarmente importante è *render*, esso dovrà obbligatoriamente essere definito in ogni Custom Component poiché restituisce un Virtual Node che descrive uno o più elementi HTML con specifiche funzionalità.
È possibile accedere alle proprietà definite sul Custom Component attraverso l'argomento *props*, attraverso quest'ultimo è possibile accedere anche alla proproietà *children* dell'elemento così da poter comporre componenti generici e maggiormente riutilizzabili [19].

Listing 3.7: Esempio di composizione

```
class Modal extends AleliComponent {
  constructor() {
    super();
    this.setState({});
  }
  render(props: VNode["props"]): VNode<{}> {
    return (
      <div className={'modal modal--' + props.type}>
        {props.children}
      </div>
   )
  }
}

class FullModal extends AleliComponent {
  constructor() {
    super();
    this.setState({});
  }
  render(props: VNode["props"]): VNode<{}> {
    return (
      <Modal type="full">
        <h1>Hello World!</h1>
      </Modal>
   )
  }
}
```

3.2.3 *Renderer e AleliRenderer*

L'interfaccia Renderer, impelementata dalla classe AleliRenderer, definisce la firma del metodo *render*, l'unico con il quale è possibile interagire con il framework.

Listing 3.8: Dichiarazione dell'interfaccia Renderer

```
interface Renderer {
  render(elementToRender: VNode, root: Element): void;
}
```

Nella classe AleliRenderer si è fatto uso del design pattern strategy, è possibile infatti invocare il costruttore passando un oggetto di tipo *Differ* e uno di tipo *RendererUtilities*, senza l'utilizzo di questo design pattern la classe *AleliRenderer* avrebbe svolto troppe attività, difficilmente testabili ed estensibili, violando i principi SOLID [15].
In questo caso la classe non conosce come verranno effettuate le operazioni, e si basa solamente sulle interfacce generiche, il che rende il framework facilmente estensibile e generico.

Listing 3.9: Dichiarazione della classe AleliRenderer implementando l'interfaccia Renderer

```
class AleliRenderer implements Renderer {
 private aleliDiffer: Differ;
 private rendererUtilities: RendererUtilities;
 private emptyVNode : VNode = { type: "", props: { children: [] } }
 // Alcuni metodi sono stati rimossi per abbreviare la porzione di
     codice
 constructor(aleliDiffer?: Differ,
  rendererUtilities?: RendererUtilities){
  if(aleliDiffer && rendererUtilities){
   this.aleliDiffer = aleliDiffer
   this.rendererUtilities = rendererUtilities;
  }
  else{
   this.rendererUtilities = new AleliRendererUtilities()
   this.aleliDiffer = new AleliDiffer(this.rendererUtilities)
  }
 }

 render(node: VNode, root: CustomHTMLElement): void {
  if (
```

```
      Array<number>(Node.TEXT_NODE, Node.COMMENT_NODE)
        .indexOf(root.nodeType) == -1
    ) {
      if(root._vnode && node.type !== root._vnode.type){
        this.removeRootDom(root)
      }
      if (!root._vnode) {
        root._vnode = this.emptyVNode;
      }
      this.aleliDiffer.diffNodes(node, root, root._vnode);
    } else {
      throw new Error(
        "AleliRenderer, can't call render method on Text or Comment root
            node"
      );
    }
  }
}
```

È infatti possibile estendere il comportamento dei moduli *Differ* e *RendererUtilities* per interagire con ambienti diversi da quello web, per esempio quello nativo desktop o mobile, l'unica condizione necessaria è descrivere gli elementi attraverso il concetto di Virtual Node.

Nel caso nessuno dei parametri venga fornito, verranno create delle istanze dei rispettivi tipi.
Il metodo *render* non può essere richiamato se l'elemento *root* rappresenta un testo o un commento, il controllo viene effettuato comparando il *nodeType* dell'elemento *root*, una proprietà numerica che ne indica il tipo, con i valori rappresentati dai tipi *TEXT_NODE* e *COMMENT_NODE*, in caso il controllo abbia esito positivo viene generata un eccezione.
Se l'elemento *root* è del tipo corretto, ma il valore della proprietà *type* è cambiato durante i render, l'intero albero di elementi nel DOM non può più essere utilizzato e deve essere scartato, un metodo performante per rimuovere l'intero albero è rimuovere il nodo principale (root) dal DOM, invece che rimuovere ogni singolo nodo.
Nel caso il metodo render venisse chiamato per la prima volta, l'elemento *root* non possiede nessuna proprietà *_vnode* e ad essa viene assegnato un VNode vuoto, infine si procede con l'operazione di diffing attraverso il metodo *diffNodes* della classe *AleliDiffer*, di ogni singolo Virtual Node che compone il Virtual DOM.

3.2.4 *RendererUtilities e AleliRendererUtilities*

Attraverso l'interfaccia *RendererUtilities*, implementata dalla classe *AleliRendererUtilities*, vengono definite le firme dei metodi utilizzati per interfacciarsi con il DOM.

Listing 3.10: Dichiarazione dell'interfaccia RendererUtilities

```
interface RendererUtilities {
  // Alcuni metodi sono stati rimossi per abbreviare la porzione di
     codice

  setProperty(
    htmlElement: CustomHTMLElement,
    prop: string,
    props: { [other: string]: any; children: Children }
  ): void

  createElement(newNode: VNode<{}>): HTMLElement | Text

  insertElementIntoDom(
    dom: CustomHTMLElement | Text,
    newNode: VNode<{}>
  ) : void

  removeProperty(htmlElement: CustomHTMLElement, prop: string) :
     void

  getOldChildren(oldNode: VNode<{}>): VNode<{}>[]
}
```

Listing 3.11: Dichiarazione della classe AleliRendererUtilities implementando l'interfaccia RendererUtilities

```
class AleliRendererUtilities implements RendererUtilities {
  // Alcuni metodi sono stati rimossi per abbreviare la porzione di
     codice

  setProperty(
    htmlElement: CustomHTMLElement,
    prop: string,
    props: { [other: string]: any; children: Children }
  ): void {
```

```
    if (!(prop in props)) {
      throw new Error(
        `Error setProperty: prop ${prop} is not present in props`
      );
    } else {
      const name: string = prop.startsWith("on") ?
          prop.toLowerCase() : prop;
      if (prop in htmlElement) {
        htmlElement[name] = props[name];
      } else {
        htmlElement.setAttribute(name, props[prop as string]);
      }
    }
  }

  insertElementIntoDom(
    dom: CustomHTMLElement | Text,
    newNode: VNode<{}>
  ): void {
    if (!("dom" in newNode)) {
      throw new Error("Can't insert element, VNode missing dom prop");
    }
    dom.insertBefore(newNode.dom as HTMLElement, null);
  }
}
```

Attraverso il metodo *setProperty* viene assegnato il valore ad uno specifico attributo dell'elemento, prima di poter effettuare questa operazione è necessario determinare se si stia trattando un *event handler* o meno.
I nomi degli *event handler*, differentemente da quelli forniti DOM, sono specificati sugli elementi HTML attraverso JSX utilizzando una nomenclatura camelCase 3.12, nel caso la proprietà inizi con le lettere *on* essa viene trasformata in minuscolo per rispecchiare le specifiche del DOM, inoltre in JSX l'*event handler* viene passato come funzione, piuttosto che stringa come invece accade in JavaScript 3.13.
Esistono due diversi approcci per registrare un *event handler*, il primo consiste nell'impostare una funzione come valore della specifica proprietà *on<event>*, sull'istanza dell'elemento HTML.

Listing 3.12: Esempio di event handler utilizzando JSX

```
class Button extends AleliComponent {
  handleClick(person: any) {
```

```
  console.log("Click");
 }

 constructor() {
  super();
  this.setState({ id: 1 });
  this.handleClick = this.handleClick.bind(this);
 }

 render(props: VNode["props"]): VNode {
  return (
   <div onClick={this.handleClick()}>
    Click Me
   </div>
  );
 }

 destroying() {}
 mounting() {}
}
```

Listing 3.13: Esempio di event handler utilizzando una funzione come valore della proprietà onclick in HTML e JavaScript

```
//File HTML
<button id="clickButton">Click Me</button>
// File Typescript
const funHandler = (e: Event) => console.log(e);
const button: HTMLButtonElement | null =
    document.getElementById("button") as HTMLButtonElement;
button.onclick = funHandler;
```

In questo caso si incorre in due limitazioni, ogni elemento può avere solamente una funzione handler per evento, e per modificare l'handler o rimuoverlo è necessario effettuare una nuova assegnazione.

Il secondo approccio consiste nell'utilizzo del metodo *addEventListener*, in questo caso ad un evento possono essere associati molteplici handler ed è possibile rimuovere ognuno di essi in caso di necessità.
Nel framework web realizzato si è scelto di utilizzare il primo approccio poiché risulta più facilmente testabile.

Facendo nuovamente riferimento al metodo *setProperty* attraverso l'operatore *in* si controlla se la proprietà è presente nell'elemento HTML, in caso il controllo si concludesse con esito positivo si starebbe trattando una proprietà standard, alla quale si può assegnare direttamente un valore; è importante specificare che le proprietà standard in JSX possono avere un nome diverso, come per la proprietà *class*, essa infatti viene specificata normalmente in un documento HTML, mentre in JSX essa prende il nome di className [20].
Nel caso il controllo abbia esito negativo, si starebbe trattando una proprietà custom, esse possono essere assegnate all'elemento HTML solamente attraverso il metodo *setProperty*.

Per l'inserimento di un nuovo elemento HTML all'interno del DOM viene utilizzato il metodo *insertElementIntoDom*, se la proprietà *dom* non è presente all'interno del VNode viene generata un'eccezione e nessun elemento sarà aggiunto, altrimenti verrà utilizzato il metodo *insertBefore* per effettuare l'aggiunta.
Si è scelto di utilizzare il metodo *insertBefore* poiché permette l'aggiunta dell'elemento in una specifica posizione, inserendolo prima dell'elemento passato come secondo parametro, in questo quest'ultimo non viene utilizzato, ma in futuro si potrebbe utilizzare aggiungendo nuove funzionalità al framework.

3.2.5 *Differ e AleliDiffer*

Attraverso l'interfaccia *Differ*, implementata dalla classe *AleliDiffer*, vengono dichiarate le firme dei metodi che compongono l'algoritmo di diffing, in particolare il metodo *findOldChildrenIfExists* avrebbe potuto essere privato, ma si è scelto di renderlo pubblico poiché la sua implementazione è fondamentale e una sua estensione permetterebbe una maggiore efficienza del framework web e l'aggiunta di nuove funzionalità.

Listing 3.14: Dichiarazione dell'interfaccia Differ

```
interface Differ{
  diffNodes(newNode: VNode,
    dom: CustomHTMLElement | Text,
    oldNode: VNode) : void

  findOldChildrenIfExists(
    oldNode: VNode<{}>,
```

```
    child: VNode<{}>,
    index: number
  ): VNode

  diffProps(
    oldVnode: VNode,
    newVnode: VNode,
    htmlElement: CustomHTMLElement
  ): void
}
```

Listing 3.15: Dichiarazione della classe AleliRendererUtilities implementando l'interfaccia RendererUtilities

```
class AleliDiffer implements Differ{
  private renderUtilities : RendererUtilities
  private detectNodeUtils : DetectNodeUtils
  // Alcuni metodi sono stati rimossi per abbreviare la porzione di
      codice

  constructor(renderUtilities : RendererUtilities, detectNodeUtils:
      DetectNodeUtils = new DetectNodeUtils()){
    this.renderUtilities = renderUtilities
    this.detectNodeUtils = detectNodeUtils
  }

  diffNodes(newNode: VNode<{}>, dom: CustomHTMLElement | Text,
      oldNode: VNode<{}>): void {
    if (typeof newNode.type !== "string") {
      this.handleDiffClassComponent(newNode,dom,oldNode)
    } else {
      this.handleDiffBaseComponent(newNode,dom,oldNode)
    }
  }

  findOldChildrenIfExists(
    oldNode: VNode<{}>,
    child: VNode<{}>,
    index: number
  ): VNode {
    const emtpyVNode: VNode = { type: "", props: { children: [] } };
    const findedVNode: VNode = this.renderUtilities
      .getOldChildren(oldNode)
```

```
    .find((oldChild) => {
      let result = undefined;
      if (
        child.props.key &&
        oldChild.props.key &&
        child.props.key === oldChild.props.key &&
        child.type === oldChild.type
      ) {
        oldChild.reusable = true;
        result = oldChild;
      } else if (child.props.key || oldChild.props.key) {
        this.markVNodeNotReusable(oldChild);
      } else {
        if (child.type === oldChild.type) {
          oldChild.reusable = true;
          result = oldChild;
        } else {
          this.markVNodeNotReusable(oldChild);
        }
      }
      return result;
    }) as VNode;

  return findedVNode || emtpyVNode;
}

diffProps(
  oldVnode: VNode,
  newVnode: VNode,
  htmlElement: CustomHTMLElement
): void {
  this.addNewProps(oldVnode,newVnode,htmlElement)
  this.removeOldProps(oldVnode,newVnode,htmlElement)
}

private destroyNotReusableComponent(notReusableVNode: VNode<{}>):
 void {
      if (notReusableVNode.component) {
        notReusableVNode.component.destroying();
        notReusableVNode.component.destroy();
      }
      this.renderUtilities.removeOldChild(notReusableVNode);
```

```
}

private addNewProps(
  oldVnode: VNode,
  newVnode: VNode,
  htmlElement: CustomHTMLElement
): void {
  const {children, key, ...props} = newVnode.props
  Object.keys(props)
  .forEach((prop) => {
    if (
      newVnode.props[prop] !== oldVnode.props[prop] ||
      newVnode.type !== oldVnode.type ||
      htmlElement.getAttribute(prop) != newVnode.props[prop]
    ) {
      this.renderUtilities.
         setProperty(htmlElement, prop, newVnode.props);
    }
  });
}

private handleDiffClassComponent(
  newNode: VNode<{}>,
  dom: CustomHTMLElement | Text,
  oldNode: VNode<{}>): void {
    if(!newNode.component){
      // @ts-ignore
      newNode.component =
          this.instantiateClassComponent(newNode.type)
    }
    if (newNode.component && !newNode.component.isMounted()) {
        newNode.component.mounting();
        newNode.component.mount();
    }
    newNode.dom = !oldNode.dom
    ? this.renderUtilities.createElement(
        newNode.component.render(newNode.props)
      )
    : oldNode.dom;
    Object.assign(oldNode, newNode);
    this.diffNodes(newNode.component.render(newNode.props), dom,
        oldNode);
```

```
  }

  private handleDiffBaseComponent(
    newNode: VNode<{}>,
    dom: CustomHTMLElement | Text,
    oldNode: VNode<{}>) : void {

      newNode.dom = !oldNode.dom
      ? this.renderUtilities.createElement(newNode)
      : oldNode.dom;
      this.renderUtilities.insertElementIntoDom(dom, newNode);

      let children: Array<VNode> = newNode.props.children as
          Array<VNode>;
      children.map((child, index) =>
          this.findOldChildAndDiff(oldNode, child, index, newNode));

      this.renderUtilities
      .getOldChildren(oldNode)
      .map(oldChild =>
           !oldChild.reusable &&
              this.destroyNotReusableComponent(oldChild))

      if (this.detectNodeUtils.isNotTextNode(newNode)) {
           this.diffProps(oldNode, newNode, newNode.dom as
              CustomHTMLElement);
      }

      Object.assign(oldNode, newNode);
  }
}
```

Attraverso il metodo *diffNodes* viene effettuato un controllo sul tipo del componente trattato attraverso l'operatore *typeof,* per gli Intrinsic Element 1.2.1 il campo *type* è di tipo stringa, differentemente invece dai Value-based Element, 1.2.2 in cui il campo *type* contiene la dichiarazione della classe del Componente o element class type [21], nel framework web esso è possibile solamente utilizzando i Class Component.
Queste due tipologie di elementi devono essere trattate diversamente nell'algoritmo di diffing.

Nel caso in cui sia trattato un Class Component è necessaria l'invocazione

del metodo *handleDiffClassComponent*, in esso si controlla la proprietà component per determinare se è già stata creata un istanza del componente altrimenti quest'ultima viene creata utilizzando la proprietà *type*, in seguito se il componente non è già stato renderizzato si procede all'invocazione dei metodi *mounting* e *mount*.

Come per gli Intrinsic Element, per poter poter inserire un elemento HTML all'interno del DOM è necessaria la sua creazione, nel caso l'elemento fosse stato creato precedentemente esso sarà accessibile attraverso la proprietà *dom*, in caso contrario la sua creazione avviene attraverso il metodo *createElement* della classe *AleliRendererUtilities*, utilizzando il VNode restituito dal metodo *render* del Custom Component. É necessario aggiornare il VNode, *oldNode*, già presente nel Virtual DOM per essere sincronizzato con le modifiche eseguite sull'elemento, per eseguire questa operazione viene usato il metodo *assign* della classe *Object*.

Il metodo *assign* effettua una copia di tutti i membri enumerabili presenti in uno o più oggetti sorgente in un oggetto destinazione, inoltre se vi sono alcune proprietà in comune, il loro valore viene aggiornato nell'oggetto destinazione, è importante specificare che non viene effettuata una deep copy, infatti in caso esistessero oggetti, come valori di chiavi, essi sarebbero solamente referenziati nell'oggetto destinazione.
L'operatore di assegnazione = avrebbe effettuato solamente una copia per riferimento, per cui *oldNode* si sarebbe riferito all'area di memoria di *newNode*.
Infine viene richiamato il metodo *diffNodes*, per effettuare l'operazione di diffing, sul VNode restituito dal metodo render, il meccanismo di diffing è ricorsivo.

Nel caso in cui il *vnode* rappresentasse un elemento HTML, il metodo *diffNodes* invoca il metodo *handleDiffBaseComponent*, anche in questo caso l'elemento HTML deve essere creato, se non lo è stato fatto nei render precedenti, per essere aggiunto al DOM.
Attraverso il metodo *findOldChildAndDiff*, descritto in seguito, verrà ricercato un Virtual Node riutilizzabile nella attuale versione del Virtual DOM, per ogni figlio del corrente VNode, e sarà eseguito nuovamente l'algoritmo di diffing.

Terminata tutta la ricorsione sui figli si passa alla rimozione dei VNode non riutilizzabili, i quali presentano la proprietà *reusable* con il valore

false, attraverso il metodo *destroyNotReusableComponent*.

Attraverso il metodo *isNotTextNode* della classe *DetectNodeUtils* si effettua un controllo sul tipo del VNode trattato, nel caso esso rappresenti un elemento di testo non sarà possibile effettuare un diffing degli attributi poiché questi elementi non possono averne. In caso contrario il metodo *diffProps* sarà invocato per aggiornare gli attributi dell'elemento al fine di riflettere quelli presenti nella nuova versione del Virtual Node, quest'ultimo metodo richiamerà i due metodi privati *addNewProps* e *removeProps*.

Nel metodo *addNewProps* si utilizza la destrutturazione del dizionario *props* contenuto nel Virtual Node, in questo modo vengono memorizzate separatamente le proprietà *key* e *children*, che non devono essere trattate, e le rimanenti proprietà sono disponibili attraverso la variabile *props*. Ogni chiave contenuta nella variabile *props* viene iterata al fine di aggiungere l'attributo, attraverso il metodo setProperty della classe *AleliRendererUtilities*, soltanto se:

- Il valore della proprietà è cambiato fra le due versioni del VNode
- Se il tipo del nodo è cambiato fra le due versioni del VNode

Infine come per i Class Component è necessario aggiornare il VNode attuale con le modifiche effettuate sull'elemento rappresentato, attraverso il metodo *assign*.
Attraverso il metodo *findOldChildrenIfExists* si effettua una ricerca nei figli del Virtual Node, *oldNode*, appartenente all'attuale versione del Virtual DOM, per trovarne uno valido 2.3.
L'array dei figli, del VNode, è ottenuto attraverso il metodo *getOldChildren* della classe *AleliRendererUtilities*, su di esso viene eseguito il metodo *find*, che restituisce il primo elemento dell'array che soddisfa la condizione espressa nella funzione test fornita e nel caso in cui nessun elemento la soddisfi sarà restituito il valore *undefined*.

La testing function è implementata utilizzando una arrow function, e in essa attraverso il primo controllo si determina se :

- Entrambi i VNode posseggono la proprietà key
- Se il valore delle proprietà key abbia lo stesso valore in entrambi i VNode

- Se il valore della proprietà type abbia lo stesso valore in entrambi i VNode

In questo caso si considera il Virtual Node riutilizzabile assegnando il valore *true* alla proprietà *reusable*.
Se solamente uno dei due nodi presenta la proprietà *key* il nodo viene automaticamente marcato come non riutilizzabile.
Nel caso nessuno dei due abbia la proprietà *key* se ne controlla il tipo, se esso risulta lo stesso il nodo viene marcato riutilizzabile altrimenti non riutilizzabile.

Un nodo può essere marcato non riutilizzabile solamente se non è stato definito riutilizzabile precedentemente, perché altrimenti quest'ultimo verrebbe distrutto se comparato in seguito con un nodo diverso.

Se il metodo *find* ha terminato la sua esecuzione senza trovare un elemento in grado di soddisfare le condizioni espresse precedentemente viene restituito il valore *undefined*, in questo caso però la funzione restituirà un VNode vuoto, segnalanado che l'elemento non esisteva precedentemente nel DOM e dovrà essere aggiunto.

3.2.6 *UML*

Attraverso il capitolo appena terminato sono stati identificate le porzioni di codice, le classi e le loro implementazioni, le funzioni di JavaScript ritenute importanti per una migliore comprensione del framework web realizzato.

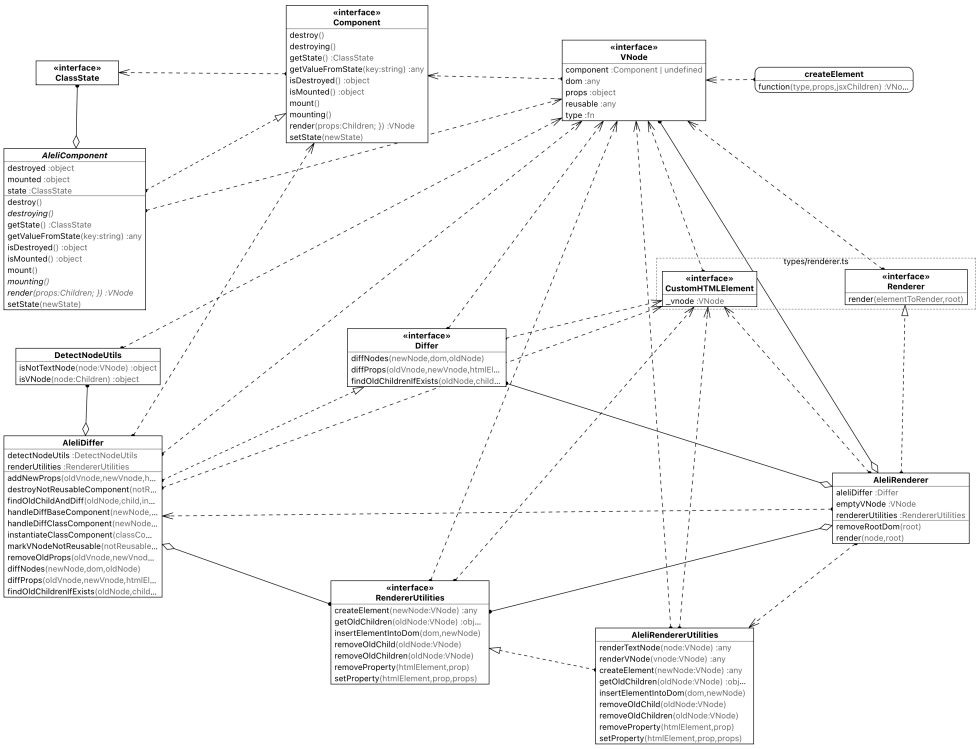

Figura 7: Diagramma UML

4

TEST EFFETTUATI

In questo capitolo verranno elencati solamente alcuni dei 152 tests realizzati tra Unit, Integration e End to End, fornendo una panoramica delle asserzioni asserzioni effettuate, quali sono gli strumenti utilizzati ed infine le problematiche riscontrate.

4.1 STRUMENTI UTILIZZATI

Nell'ecosistema JavaScript e TypeScript il framework più utilizzato per il testing è Jest.
Quest'ultimo risulta compatibile con TypeScript attraverso l'utilizzo del compilatore Babel e l'aggiunta di specifici preset come *@babel/preset-typescript*, in questo caso viene meno la funzionalità di type checking, poiché attraverso Babel tutte le informazioni sui tipi verranno rimosse.
Per poter sfruttare a pieno le potenzialità di type checking è necessario l'utilizzo di uno specifico transformer Jest, nel progetto di laurea discusso si è utilizzato ts-jest.
Jest mette a disposizione la funzione *expect*, utilizzata ogniqualvolta sia necessario effettuare un asserzione, comunemente invocata utilizzando uno specifico matcher, ad esempio *toBe*, *toBeCalled*, *toBeInstanceOf*.

Per ogni categoria di test effettuati è stato necessario fornire specifici parametri attraverso la creazione di molteplici file di configurazione per Jest, in questo modo è stata facilitata anche la loro esecuzione attraverso il CI Server.
È stato utilizzato il pacchetto ts-mockito per poter effettuare operazioni di mocking e spying più facilmente rispetto a Jest, in cui esse risultano lunghe e complesse.

4.2 UNIT TEST

4.2.1 *createElement*

Listing 4.1: Unit test per la funzione createElement

```
it("createElement return VNode where props contain specified attribute ",
    () => {
   const expectedVNode: VNode <{id: "Aleli"}> =
    {
      type: "div",
      props: { children: ["Hello"], id: "Aleli" },
    };
   expect(createElement("div", {id: "Aleli", children: []}, "Hello"))
    .toHaveProperty('props.id',"Aleli");
 });

 it("createElement return VNode where children is array that contain
     VNode", () => {
   const expectedVNode: VNode = {
    type: "div",
    props: {
     children: [{
        type: "span",
        props: {
         children: [{
            type: "$TEXT",
            props: { textValue: "Hello", children: []}
          }],
        },
       }],
    },
   };
   expect(
    createElement("div", {children: []}, createElement("span",
        {children: []}, "Hello"))
   ).toEqual(expectedVNode);
 });
});
```

Questi test coprono alcuni casi d'uso della funzione *createElement,* in essi ci si assicura che la funzione restituisca il Virtual Node corretto in base ai

parametri ricevuti in input.

4.2.2 *Component*

Per effettuare i test sui component è necessario effettuare delle operazioni di mocking, per convenzione di Jest i file contenenti un mock devono essere definiti all'interno della cartella *__mocks__*,

Listing 4.2: Unit test per Class Component

```
import TestComponent from "../../__mocks__/testComponent.mock";

it('AleliComponent should have intitial state empty', () => {
   const state : Object = testComponent.getState();
   expect(Object.keys(state).length).toBe(0)
 });

it('AleliComponent should read value of key using getValueFromKey',
  () => {
       testComponent.setState({id:1})
       const valueOfKey : any =
           testComponent.getValueFromState('id');
       expect(valueOfKey).toBe(1)
  })

it('AleliComponent getValueFromKey method should return undefined if
    store doesn't contain specified key', () => {
   const valueOfKey : any = testComponent.getValueFromState('Hello');
   expect(valueOfKey).toBe(undefined)
  });
```

Nei test elencati vengono effettuate delle asserzioni sul valore assunto dallo state, nel primo test ci si aspetta che dopo aver istanziato il componente, di cui è stato creato un mock, lo state sia vuoto.
Nei casi successivi si controlla il valore restituito dalla funzione *getValueFromState*, ad essa viene passata la chiave con cui accedere al dizionario, nel caso la chiave non sia presente sarà restituito il valore *undefined*.

4.2.3 *AleliRenderer*

Per poter testare il corretto funzionamento della classe *AleliRenderer* è necessario creare i mock, e le istanze ottenute da quest'ultimi, delle

due dipendenze *AleliDiffer* e *AleliRendererUtilities*. Esse vengono create attraverso le funzioni *mock* ed *instance*, invocate nel metodo *beforeAll*, forniti dal pacchetto ts-mockito.

Listing 4.3: Unit test per AleliRenderer

```
beforeAll(() => {
   mockAleliDiffer = mock(AleliDiffer)
   instanceAleliDiffer = instance(mockAleliDiffer)
   mockRendererUtilities = mock(AleliRendererUtilities)
   instanceRendererUtilities = instance(mockRendererUtilities)
   aleliRenderer = new AleliRenderer(instanceAleliDiffer,
       instanceRendererUtilities)
   spiedAleliRenderer = spy(aleliRenderer)
 })

it('AleliRenderer render method remove root dom element if type between
    re-renders diff',
  () => {
    const rootElement : CustomHTMLElement =
      document.createElement("div")
    const spanElement : CustomHTMLElement =
      document.createElement("span")
    const spiedSpanElement: CustomHTMLElement =
      spy(spanElement)

    rootElement._vnode = {
        type: 'span',
        props: {
          children: []
        },
        dom: spanElement
    }

    const vnode : VNode = {
      type: "div",
        props: {
          children: []
      }
    }

    aleliRenderer.render(vnode,rootElement)
    verify(spiedSpanElement.remove()).once()
```

```
});
```

Attraverso il test ci si assicura che sia invocata la funzione remove, dell'elemento html corrispondente al nodo root del Virtual Dom, in caso di modifica della proprietà *type* di quest'ultimo Per poter verificare l'invocazione della funzione si effettua uno *spy* sull'elemento HTML attraverso l'omonima funzione anch'essa fornita dal pacchetto ts-mockito.

4.2.4 *AleliDiffer*

I test realizzati per lo sviluppo della classe *AleliDiffer* sono i più numerosi, come descritto nei capitoli precedenti questa classe svolge un ruolo fondamentale all'interno del framework e come per la classe *AleliRenderer* è necessaria la creazione dei mock e delle relative istanze.
Vengono elencati i test della suite per il metodo *findOldChildrenIfExists*

Listing 4.4: Unit test per la funzione findOldChildrenIfExists della classe AleliDiffer

```
it("aleliDiffer method findOldChildrenIfExists, if keys are used, not reusable
    vnode are flagged", () => {

  const oldVNode: VNode = {
    type: "div",
    props: {
      key: 1,
      children: [],
    },
  };

  const oldVNodeSecond: VNode = {
    type: "span",
    props: {
      key: 2,
      children: [],
    },
  };

  const vnode: VNode = {
    type: "div",
    props: {
      children: [oldVNode, oldVNodeSecond],
    },
```

```
    };

    const newChildVNode: VNode = {
      type: "span",
      props: {
        key: 2,
        children: [],
      },
    };

    when(mockedRendererUtilities.getOldChildren(vnode)).
      thenReturn([
          oldVNode,
          oldVNodeSecond,
      ]);

    aleliDiffer.findOldChildrenIfExists(vnode, newChildVNode, 0)
    expect(oldVNode.reusable).toBe(false)
    expect(oldVNodeSecond.reusable).toBe(true)
  });

  it("differProp should'nt call setProperty for key prop", () => {
    const oldVnode: VNode = {
      type: "div",
      props: {
        key: 2,
        children: [],
      },
    };
    const newVnode: VNode = {
      type: "div",
      props: {
        key: 1,
        children: [],
      },
    };
    const htmlElement: HTMLElement =
      document.createElement("div");
    aleliDiffer.diffProps(oldVnode, newVnode, htmlElement);
    verify(
      mockedRendererUtilities.
        setProperty(htmlElement, "key", newVnode.props)
```

```
    ).never();
  });
```

Attraverso il primo test elencato, ci si assicura che i Virtual Node presi in considerazione dall'algoritmo di diffing, al suo termine, posseggano la proprietà *reusable* con il corretto valore.
Quando il metodo *findOldChildrenIfExist* viene invocato, l'algoritmo di diffing compara *newChildVNode* con tutti i Virtual Node che formano l'array *children*, solamente *oldVNodeSecond* risulta riutilizzabile perché nei controlli effettuati le proprietà *key* e *type* coincidono.
Nel secondo test viene effettuata un asserzione sul numero di invocazioni alla funzione *setProperty* messa a disposizione dalla classe *AleliRendererUtilities*, come specificato nel precedente capitolo 3.2.5 le proprietà speciali *key* e *children* non devono essere assegnate agli elementi HTML.

4.3 INTEGRATION TESTS

Gli integration test sono stati realizzati fra i componenti che cooperano durante l'esecuzione dell'applicativo, come ad esempio JSX e la funzione *createElement* o come *AleliRenderer* e le classi *AleliRendererUtilities* e AleliDiffer. Di seguito viene elencato un ristretto numero di integration tests con lo scopo di descrivere le asserzioni effettuate.

Listing 4.5: Integration Tests JSX e factory function

```
it("JSX call createElement, return vnode with prop", () => {
    const vnodeAleliElement: VNode = {
      type: "div",
      props: {
        id: 1,
        className: "box",
        children: [],
      },
    };
    const aleliDivElement: HTMLDivElement =
      <div id={1} className="box"></div>;
    expect(aleliDivElement).toStrictEqual(vnodeAleliElement);
  });

it("JSX call createElement, return vnode with class component", () => {
    const vnodeAleliElementProps: VNode["props"] = {
      id: "1",
```

```
    testProp: false,
    children: [],
  };
  const aleliDivElement: VNode = (
    <AleliComponent id="1" testProp={false}></AleliComponent>
  );
  expect(aleliDivElement.type).toBe(AleliComponent as Function);
  expect(aleliDivElement.props).toStrictEqual(vnodeAleliElementProps);
});
```

Nei test sopra elencati vengono effettuate asserzioni sul Virtual Node restituito dalla funzione *createElement*, che viene implicitamente invocata attraverso il pre-processore quando un Token JSX viene valutato. Nel caso in cui venga utilizzato un Class Component ci si aspetta che il tipo del valore della proprietà *type* corrisponda ad una funzione e non ad una stringa.
Questa è la prima volta in cui la factory function e JSX interagiscono, per cui la validità di questi test è fondamentale per aver confidenza sui possibili cambiamenti alla *createElement* possibilmente necessari in futuro.

Listing 4.6: Integration Test AleliRender e JSX

```
it('AleliDiffer diffNodes method update dom and set onClick listener',
  () => {
      const fun = () => 3
      const oldVNode: VNode = <div></div>
      const newVNode: VNode = <div onClick={fun} ></div>
      const rootElement : HTMLDivElement =
        document.createElement("div")
      const divElement: HTMLDivElement =
        document.createElement("div")
      aleliDiffer.diffNodes(newVNode,rootElement,oldVNode)
      const htmlElement: HTMLElement =
        rootElement.firstChild as HTMLElement
      expect(typeof htmlElement.onclick).toBe("function")
  })
```

In questo integration test la classe *AleliDiffer* interagisce con JSX e *createElement*, si esegue un asserzione sull'attributo *onclick* del *div* renderizzato. L'elemento rimane valido durante il re-render, in questo caso simulato attraverso l'invocazione del metodo *diffNodes*, ma le proprietà differiscono poiché nella nuova versione del Virtual Node è presente la proprietà *onclick*. L'asserzione assicura che il tipo di quest'ultimo attributo sia *func-*

tion e implicitamente che quest'ultimo sia stato assegnato all'elemento dall'algoritmo di diffing.

4.4 END TO END TEST

Nei test E2E ci si avvicina molto al caso d'uso di un reale applicativo web, in esso vengono definiti solamente Class Component che possono utilizzare il concetto di composizione 3.7, implementano funzioni per la gestione di eventi 3.12, fanno uso dei metodi che compongono il ciclo di vita del componente 6, ed utilizzano lo state.

Listing 4.7: End to end Test

```
it("Aleli can re render application entirely", () => {
   const hello : string = "Hello Aleli!"
   const helloComponent = (<div>{hello}</div>)
   aleliRenderer.render(helloComponent,root)
   const child : HTMLElement = root.firstChild as HTMLElement
   expect(child.textContent).toEqual(hello)
   const component =
       (<span><HelloWorldComponent></HelloWorldComponent></span>)
   aleliRenderer.render(component,root)
   const childUpdated : HTMLElement = root.firstChild as HTMLElement
   expect(childUpdated.textContent).toEqual("Hello World Aleli with
       Class Component!")
})

it("Aleli can render class component with click listener", async (done) =>
    {
   const handleClick = (e: Event) => {
      done()
   }
   const component = (<ClickComponent handleClick={handleClick} />)
   aleliRenderer.render(component,root)
   const child : HTMLElement = root.firstChild as HTMLElement
   await fireEvent.click(child.firstChild as HTMLElement)
})

it("Aleli can render class component with state", () => {
   const component = (<HelloWorldComponent></HelloWorldComponent>)
   aleliRenderer.render(component,root)
   const child : HTMLElement = root.firstChild as HTMLElement
   expect(child.id).toEqual("1")
```

```
    expect(child.textContent).toEqual("Hello World Aleli with Class
        Component!")
})
```

Nel primo test End to End ci si assicura che gli elementi visualizzati nella pagina web siano corretti, se come risultato del primo *render* si ha un elemento *div* contenente al suo interno la stringa "Hello Aleli!", il secondo *render* provoca un aggiornamento degli elementi presenti senza un ri-caricamento dell'intera pagina.
La pagina dunque conterrà solamente un elemento *span* con al suo interno la stringa "Hello World Aleli with Class Component!"
Attraverso il secondo test ci si assicura che gli event handler funzionino correttamente se un evento è stato innescato, nel nostro contesto al *ClickComponent* viene passata come *props* una funzione che rappresenterà l'event handler per l'evento click. L'evento viene innescato attraverso *fireEvent.click(element)* una funzione messa a disposizione dal pacchetto testing-library.
Questo test non presenta nessuna asserzione esplicita poiché essa viene effettuata implicitamente invocando la funzione *done()*, quest'ultima è una funzione passata come argomento al test.
Nell'ultimo test si effettua anche un asserzione sull'utilizzo dello stato, *HelloWorldComponent* renderizza un elemento HTML *div* con il valore dell'attributo *id* che deriva dallo stato del componente stesso.
Lo stato viene inizializzato all'interno del metodo *mounting* del componente e sarà disponibile nella fase di render.

Attraverso quest'ultimo capitolo sono stati presentati un ristretto numero dei test realizzati per poter rappresentare le metodologie di testing utilizzate.
È necessaria una riflessione su JavaScript e TypeScript, seppur siano linguaggi molto utilizzati e con un vasto supporto della community, la mancanza di una dettagliata documentazione o il non corretto funzionamento di alcune dipendenze hanno causato diversi problemi nel corso dello sviluppo del progetto di laurea, ed ha causato molteplici blocchi e ricerche di pacchetti funzionanti alternativi.

CONCLUSIONI

Lo sviluppo del framework web seppur in versione prototipale è risultata un'operazione complessa con molteplici sfide da affrontare e risolvere. La mancanza di bibliografia sullo sviluppo di framework, tolti alcune risorse presenti online, ha portato alla consultazione di numerose pagine web le cui informazioni non erano sempre appropriate o che proponevano soluzioni non ottimali e non utilizzabili, è stato quindi di fondamentale importanza lo studio di alcune codebase di framework preesistenti.
Proprio la consultazione di codebase, anche di grande dimensione e elevata complessità, ha permesso di assimilare nuovi concetti e una panoramica più generale sull'attuale stato dello sviluppo web. Il Testing di componenti e applicativi web ha dimostrato che tutt'ora, secondo il mio personale parere, i tools e la documentazione disponibili non risultano essere ottimali o fruibili come dovrebbero, e la svariata disponibilità di pacchetti non testati eo non più mantenuti crea uno stato di stallo, in conclusione la scrittura è risultata più difficoltosa rispetto a quella incontrata ad esempio con il linguaggio Java.
In conclusione attraverso lo sviluppo di questo progetto di laurea si sono apprese nuove nozioni che risultano essere fondamentali anche in ambito lavorativo.

BIBLIOGRAFIA

[1] Nominal and structural typing. https://www.eclipse.org/n4js/features/nominal-and-structural-typing.html. (Cited on page 9.)

[2] Wikipedia. Strong and weak typing. https://en.wikipedia.org/wiki/Strong_and_weak_typing. (Cited on page 9.)

[3] Jsx: Xml-like syntax extension to ecmascript. https://facebook.github.io/jsx/. (Cited on page 14.)

[4] Dom 3 specification. https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.html. (Cited on page 15.)

[5] Mark Wilton-Jones. Efficient javascript. https://dev.opera.com/articles/efficient-javascript, Nov 2006. (Cited on page 16.)

[6] Lindsey Simon. Minimizing browser reflow. https://developers.google.com/speed/docs/insights/browser-reflow. (Cited on page 16.)

[7] Mattias Levlin. Dom benchmark comparison of the front-end javascript frameworks react, angular, vue, and svelte. *Levlin, Mattias*, 2020. (Cited on page 21.)

[8] Wikipedia. Test-driven development — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Test-driven%20development&oldid=1021946301, 2021. (Cited on page 22.)

[9] Bettini Lorenzo. *Test-Driven Development, Build Automation, Continuous Integration*. Lorenzo Bettini, 1 edition, 2021. (Cited on pages 23 and 24.)

[10] Robert C. Martin. The three laws of tdd. http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd. (Cited on page 23.)

[11] Fowler Martin. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html, 2006. (Cited on page 24.)

[12] Reactjs - riconciliazione. https://it.reactjs.org/docs/components-and-props.html. (Cited on page 30.)

[13] Philip Bille. A survey on tree edit distance and related problems. https://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf. (Cited on page 30.)

[14] HTTP Archive. Http archive: State of the web. https://httparchive.org/reports/state-of-the-web. (Cited on page 30.)

[15] Wikipedia. Solid principles. http://it.wikipedia.org/w/index.php?title=SOLID&oldid=118748001. (Cited on pages 33 and 43.)

[16] Microsoft. Typescript: Documentation - declaration merging. https://www.typescriptlang.org/docs/handbook/declaration-merging.html. (Cited on page 37.)

[17] Mozilla MDN. Function apply - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply?retiredLocale=it. (Cited on page 40.)

[18] Mozilla MDN. Array function concat - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?retiredLocale=it. (Cited on page 40.)

[19] Facebook ReactJs. Composizione e ereditarità – react. https://it.reactjs.org/docs/composition-vs-inheritance.html. (Cited on page 42.)

[20] Quora. classname and class. https://www.quora.com/Why-do-I-have-to-use-className-instead-of-class-in-ReactJs-components\-done-in-JSX-JSX-is-preprocessed-so-shouldnt-that-conversion-happen-when-\JSX-is-converted-to-JavaScript. (Cited on page 48.)

[21] Microsoft. Typescript: Documentation - class component. https://www.typescriptlang.org/docs/handbook/jsx.html#class-component. (Cited on page 52.)