Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea in Informatica

Il data binding nelle web app: un'analisi comparativa

Relatore:
Chiar.mo Prof. Fabio Vitali

Presentata da:
Nicole Sabrina Marro

II Sessione
Anno Accademico 2024/2025

To all the imposters out there: YOU did it.

Abstract

Il presente elaborato si pone l'obiettivo di analizzare il *data binding*, un meccanismo fondamentale che si trova alla base della sincronizzazione tra modello e vista nelle applicazioni web moderne. Dopo aver definito le principali tipologie e problematiche del binding, la tesi procede analizzando nel dettaglio le diverse modalità di implementazione del binding all'interno dei seguenti framework: *Angular*, *React*, *Vue* e *Svelte*. L'analisi è condotta attraverso paragoni su problematiche comuni ed esempi di codice esplicativi ed intuitivi. Successivamente viene presentata una valutazione comparativa tra i vari modi di fare binding dei suddetti framework, in base a criteri di sintassi, espressività ed efficienza. L'analisi mette in evidenza i punti di forza e le limitazioni di ciascun paradigma, delineando le loro tendenze evolutive verso forme di reattività più predittive e compile-time e mettendole a confronto con le più nuove tecnologie introdotte nel campo. L'obiettivo finale è quello di fornire una visione sistematica delle soluzioni esistenti e di delineare le tendenze evolutive che caratterizzeranno la discussione sul binding negli anni a venire.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

L'esperienza utente è uno dei punti cardine attorno al quale ruota il mondo dello sviluppo web, in quanto essa fa parte dei metri di valutazione per decidere se ci troviamo di fronte ad una buona applicazione o meno. Per tale motivo, negli ultimi anni, l'esigenza di fornire interfacce sempre più reattive, fluide e coerenti ha portato ad una grande evoluzione nello sviluppo di applicazioni web. In questo contesto, il *data binding* è diventato uno dei fulcri della progettazione dei framework moderni, poiché è il meccanismo che rende possibile la sincronizzazione tra modello e vista e che riduce la complessità di gestione dello stato, favorendo la coerenza dell'applicazione.

Il binding, che collega i dati del modello logico con la loro rappresentazione nell'interfaccia, ha assunto nel tempo forme e caratteristiche diverse, rispecchiando l'evoluzione delle esigenze di efficienza e scalabilità del software. Se alcuni framework si concentrano sul favorire un'interazione immediata e bidirezionale tra utente e modello, altri preferiscono approcci più controllati e predittivi, basati su flussi di dati unidirezionali o sulla generazione di logiche reattive già in fase di compilazione.

Questa tesi si propone di analizzare in modo comparativo la sintassi, l'espressività e l'efficienza dei diversi approcci al data binding nei framework JavaScript/TypeScript utilizzati maggiormente nello sviluppo web contemporaneo, con l'obiettivo di comprendere i punti di forza, le limitazioni e le tendenze evolutive di ciascun paradigma. L'idea di fondo di questo elaborato nasce, inoltre, dalla volontà di cogliere le novità proposte recentemente dalla comunità di sviluppatori e ricercatori del

settore, che da anni riflettono su quale sia la soluzione più bilanciata tra prestazioni, predicibilità e semplicità d'uso.

Nel primo capitolo introdurremo il concetto di *data binding*, presentando le tipologie principali (one-way, two-way e one-time) e i meccanismi di osservazione che i framework adoperano per rilevare i cambiamenti nello stato dell'applicazione. Saranno inoltre affrontate le problematiche moderne legate a efficienza, correttezza, scalabilità e sicurezza, che costituiscono le basi teoriche per il confronto successivo.

Successivamente, nel secondo capitolo, approfondiremo le strategie differenti che i principali framework (*Angular*, *React*, *Vue* e *Svelte*) adottano per implementare il data binding, evidenziando similarità, differenze e punti di forza e di debolezza di ognuno. L'analisi comprende aspetti sintattici, architetturali e prestazionali, supportati da esempi di codice pratici e da una matrice comparativa che mette in relazione criteri di efficienza, espressività e semplicità d'uso.

Il terzo e ultimo capitolo della dissertazione amplierà la prospettiva verso le nuove tendenze tecnologiche, illustrando sia le direzioni intraprese dai framework analizzati nel capitolo precedente, sia le tecniche introdotte da quelli contemporanei come Qwik e Solid.js e le più recenti innovazioni in ambito reattivo.

Infine, attraverso questa analisi, giungeremo a fornire un quadro chiaro e sistematico sull'evoluzione del data binding nel web development, mettendo in luce come la scelta del framework incida sulle prestazioni, sulla leggibilità del codice e sulla capacità di realizzare applicazioni scalabili, in modo da poter decidere più consapevolmente quale framework utilizzare in contesti con esigenze differenti.

In sintesi, il lavoro mira a contribuire alla riflessione critica sul binding come problema ancora aperto e centrale nello sviluppo moderno, evidenziando come ogni framework rappresenti una diversa risposta all'obiettivo comune di rendere la comunicazione tra dati e interfaccia più efficiente, espressiva e predicibile.

Capitolo 2

Il concetto di Data Binding

2.1 Definizione di data binding

Nel contesto dello sviluppo di applicazioni web moderne, il *data binding* rappresenta il processo che connette l'interfaccia utente con i dati sottostanti che essa espone. In questo meccanismo, l'UI rappresenta la *view* mentre i dati rappresentano il *model*, dunque il binding e' ciò che permette la sincronizzazione tra lo stato interno dell'applicazione e la sua rappresentazione esterna. In termini semplici, esso garantisce che ogni modifica ad uno tra model e view si rifletta sull'altro [Mic25].

Il data binding non assicura solo la coerenza del flusso informativo tra model e view, ma anche tra gli elementi figli del modello padre. In pratica, propaga le modifiche in tutte le parti in cui i dati vengono usati e mostrati sull'interfaccia. Ad esempio, se io ho un ricettario (il *model*) e un menu (la *view*) con al suo interno vari piatti (i figli) e decido di cambiare delle direttive del ricettario, automaticamente vengono aggiornati anche il menu e tutti i suoi piatti figli.

2.1.1 Pattern architetturali: MVC e MVVM

Il concetto di data binding si inserisce storicamente all'interno di alcuni *design pattern* che hanno guidato lo sviluppo delle UI, in particolare di due modelli: **MVC** (Model–View–Controller) e **MVVM** (Model–View–ViewModel).

Il pattern **MVC**, nato negli anni '70, suddivide l'applicazione in tre componenti principali: il *Model*, ovvero i dati e la logica di business; la *View*, che si occupa della presentazione grafica; e il *Controller*, una sorta di mediatore tra le due, che riceve gli input dell'utente e aggiorna il modello e la vista di conseguenza. Il pregio di questo approccio è quello di separare le responsabilità, ma risulta sprovvisto di un meccanismo automatico di sincronizzazione tra modello e vista, dunque tale compito rimane a carico del controller.

Il pattern **MVVM**, introdotto successivamente e diffuso in numerosi framework JavaScript moderni, rappresenta un'evoluzione di MVC orientata alle interfacce dichiarative. In esso, la *ViewModel* agisce come ponte tra modello e vista ed espone proprietà e comandi legati (*bound*) direttamente nella view. In questo caso, è il data binding a rappresentare il meccanismo cardine che permette alla view di riflettere automaticamente i cambiamenti del model e viceversa, dunque viene notevolmente ridotto o eliminato del tutto il ruolo del controller. È proprio il modello MVVM ad aver favorito la diffusione del two-way binding nei framework web, a partire da AngularJS fino a Vue e Angular moderno.

2.1.2 Direzioni del binding

A seconda della direzione della sincronizzazione, si distinguono principalmente tre modalità:

- **one-way binding**, in cui il flusso dei dati è unidirezionale dal modello verso la vista;
- **two-way binding**, in cui anche le interazioni dell'utente sulla vista aggiornano automaticamente il modello;
- **one-time o lazy binding**, in cui il legame tra modello e vista è stabilito una sola volta, senza aggiornamenti successivi.

Analizzeremo queste tre tipologie nella sezione 2.2, costruendo una base teorica necessaria per comprendere come i diversi framework le implementino con strategie differenti.

2.2 Tipologie di data binding

Le tre forme di data-binding si distinguono principalmente per la direzione in cui avviene il flusso di aggiornamento di dati, per implicazioni architetturali e soprattutto per il loro casi d'uso.

2.2.1 One-way binding

Nel *one-way binding* i dati fluiscono dal modello (sorgente di verità) verso la vista, che ne è una proiezione, ma non viceversa, dunque ogni dato può essere passato dal componente padre a tutti i figli automaticamente, ma per implementare il flusso contrario vi è la necessità di dichiararne la gestione manualmente. Viene notoriamente impiegato dal framework React quando si utilizza l'hook useState per definire le variabili (flusso unidirezionale *state* → *props*), che approfondiremo più avanti. Di seguito vediamo un piccolo esempio di binding monodirezionale in JavaScript:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Esempio One-way Binding</title>
</head>
<body>
  <!-- VIEW: l'elemento da aggiornare -->
  <div id="elementoDaAggiornare"></div>

  <script>
    // MODEL
    const model = { price: 100 };

    // RENDERING DELLA VIEW
    function renderFunction() {
      document.getElementById('elementoDaAggiornare').textContent =
        `Prezzo: ${model.price} €`;
```

```
    }

    // Aggiorno solo il modello, poi aggiorno la vista
    model.price = 220;
    renderFunction();
  </script>
</body>
</html>
```

Questo approccio è chiaramente molto semplice ed è usato soprattutto per la prevedibilità del flusso e per la tracciabilità degli aggiornamenti. Infatti, come si vede nel codice, la vista è aggiornata esplicitamente a partire dal modello: l'utente non modifica direttamente lo stato, evitando comportamenti inaspettati e gap nella performance. Di contro si pone il problema di dover dichiarare esplicitamente tutti i setter per far si che anche le modifiche della view possano propagarsi sul modello, generando uno pseudo binding bidirezionale più verboso [Vou17].

2.2.2 Two-way binding

Nel *two-way binding* cambiamenti nel modello si riflettono sulla view e modifiche su una view si riflettono a loro volta sul modello e su tutte le altre view dipendenti, quindi l'input dell'utente nella vista aggiorna automaticamente il modello. È molto comodo per gestire form e controlli di input, come ad esempio nei riepiloghi degli acquisti su e-commerce in cui ogni aggiunta ed eliminazione deve propagarsi nel prezzo totale, nella percentuale di sconto applicata etc. Framework che usano questo modello sono: Angular con sintassi `[(ngModel)]` ([Koc23]); Vue con direttiva `v-model` ([Vue25c]); Svelte con direttiva `bind:`([Sve25e]). Di seguito vediamo un piccolo esempio di binding bidirezionale in javascript:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
```

```
  <title>Esempio Two-way Binding</title>
</head>
<body>
  <h3>Two-way binding</h3>

  <!-- Vista: un input e un div -->
  <input id="inputElement" type="text">
  <div id="outputElement"></div>

  <script>
    // MODEL: i dati
    const model = { testo: "ciao" };

    const input = document.getElementById("inputElement");
    const output   = document.getElementById("outputElement");

    // MODEL -> VIEW (aggiorna la UI quando cambia il modello)
    function funzioneDiRender() {
      input.value = model.testo;
      output.textContent = "Testo: " + model.testo;
    }

    // VIEW -> MODEL (aggiorna il modello quando cambia la UI)
    input.addEventListener("input", (e) => {
      model.testo = e.target.value;  // aggiorna il modello
      funzioneDiRender();   // propaga il cambiamento
    });

    // Prima visualizzazione
    funzioneDiRender();
  </script>
</body>
```

```
</html>
```

In questo esempio vediamo un approccio più intuitivo e snello che viene impiegato soprattutto nella gestione dei form: riduce la quantità di codice necessario per tenere aggiornati bidirezionalmente view e model. Tuttavia, a causa di ciò vi è una maggiore complessità nel tracciamento del flusso dei dati, nel quale si possono generare aggiornamenti indesiderati o addirittura ciclici se non vengono gestite accuratamente le dipendenze, cosa che può compromettere la prevedibilità in applicazioni di grandi dimensioni.

2.2.3 One-time binding

Nel *one-time binding*, infine, la vista legge il modello una sola volta in fase di inizializzazione, dopodiché non partecipa ulteriormente alle modifiche ad esso apportate. Risulta utile per contenuti statici o che non richiedono di essere modificati (es. etichette, descrizioni) e per ottimizzare la performance dell'app eliminando controlli inutili sugli elementi che devono rimanere immutabili. Nella pratica, lo vediamo in funzione in framework quali Vue, che espone la direttiva `v-once` per rendere un elemento (e i suoi figli) statici dopo il primo render [Vue25a] e AngularJS, che ha introdotto l'operatore `::` per espressioni valutate una tantum ([Ang]). Di seguito un esempio di one-time binding:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Esempio One-time Binding</title>
</head>
<body>
  <h3>One-time binding</h3>

  <!-- Vista: un div statico -->
  <div id="outputElement"></div>
```

```
  <button id="updateButton">Cambia modello</button>

  <script>
    // MODEL
    const model = { titolo: "Starting version" };

    const output = document.getElementById("outputElement");
    const button = document.getElementById("updateButton");

    // ONE-TIME BINDING: copia il valore una sola volta
    output.textContent = model.titolo;

    // Cambiamento del modello
    button.addEventListener("click", () => {
      model.titolo = "New version";
      alert("Il modello è diventato: " + model.titolo);
      // La vista non si aggiorna
    });
  </script>
</body>
</html>
```

Questo schema, dunque, fornisce un modo per liberare risorse dopo che è avvenuto il binding, riducendo il numero di espressioni che necessitano di controlli e aggiornamenti continui e rendendo più veloce la renderizzazione e la reattività dell'interfaccia.

Avendo analizzato le varie tipologie di binding, sorge spontaneo domandarsi in che modo i framework operino per rendersi conto dei cambiamenti avvenuti in model e view e poterli propagare correttamente. Tali meccanismi sono i cosiddetti *meccanismi di osservazione*, oggetto della prossima sezione.

2.3 Meccanismi di osservazione

Il modo in cui i framework si accorgono che un dato è cambiato e decidono, quindi, come e quando aggiornare la vista è un aspetto fondamentale del data binding che ha a che fare con la *reattività* delle applicazioni, uno dei protagonisti del web developing. Per sviluppare una buona app user friendly, spesso ci si concentra sulle modalità e sulla velocità di reazione dei componenti alle interazioni con l'utente, poiché queste garantiscono un'esperienza più o meno positiva. Questo compito è svolto dai *meccanismi di osservazione*, che possono essere classificati in tre approcci principali: *pull/diff*, *push/reattivo* e *compile-time*.

2.3.1 Pull / diff

Nell'approccio *pull* il framework confronta periodicamente lo stato attuale del modello con quello precedente per capire se è necessario aggiornare la vista. Questo schema era adottato ad esempio da AngularJS, in cui è noto come *dirty checking* poiché caratterizzato da un semplice ciclo di digest che verifica le variazioni osservate sulle espressioni e da React e Vue, in cui è noto come *diffing* in quanto fanno uso di un Virtual DOM, cioè di una copia ad albero del DOM con la quale si confronta il DOM attuale per accorgersi delle modifiche [Wik25a; Vue25f].

È un approccio concettualmente molto semplice, tuttavia ha lo svantaggio di dover eseguire controlli continui anche in assenza di modifiche, con potenziali sprechi di risorse.

2.3.2 Push / reattivo

L'approccio *push*, invece, funziona esattamente in maniera inversa: è il modello stesso a notificare immediatamente i cambiamenti agli osservatori interessati. Questa logica si basa su *signals*, *observables* o sistemi di dependency tracking, in cui la vista si aggiorna solo dove serve. Framework come Vue 3 [Vue25e], Solid.js e le versioni più recenti di Angular adottano questo modello [Bui24; Ang25b], servendosi di costrutti quali Proxy o signals, che vedremo in seguito.

Questo schema è più efficiente perché aggiorna solo i nodi dipendenti, ma comporta anche la necessità di gestire correttamente il ciclo di vita delle sottoscrizioni: se un componente viene rimosso dall'interfaccia senza che le sue sottoscrizioni vengano cancellate, esse rimangono attive, potenzialmente generando *memory leak*.

2.3.3 Compile-time

Infine, alcuni framework spostano il problema della reattività ai cambiamenti dal runtime al momento della compilazione dei template. In questo caso, il compilatore genera direttamente le istruzioni di aggiornamento necessarie per aggiornare le dipendenze, poiche queste vengono determinate staticamente. Tutto ciò elimina la necessità di confronti o di notifiche a runtime ed è l'approccio scelto ad esempio da Svelte, che riduce al minimo l'overhead durante l'esecuzione [Kle23].

Tale meccanismo garantisce aggiornamenti rapidi e precisi poiché non serve piú logica aggiuntiva, ma va a scapito della flessibilità: eventuali pattern molto complessi e dinamici non prevedibili a build time possono risultare piú difficili da gestire e richiedere soluzioni meno immediate rispetto ai modelli elencati sopra.

Di seguito è disponibile una tabella riassuntiva della sezione a cui faremo riferimento in seguito per accesso facilitato e leggibilità più diretta.

Tabella 2.1: Confronto tra i principali meccanismi di osservazione

Approccio	Pro	Contro
Pull / diff	Più semplice; no codice aggiuntivo richiesto allo sviluppatore; reso popolare da AngularJS (dirty checking) e da React/Vue (Virtual DOM diffing).	Necessitá di ricontrollare continuamente anche in assenza di cambiamenti; spreca potenzialmente risorse; aggiornamenti spesso più grossolani (coarse-grained).
Push / reattivo	Aggiornamenti mirati; molto efficiente se si hanno molti nodi e poche modifiche; utilizzato da Vue 3 (Proxy), Angular Signals e Solid.js.	Richiede attenzione per gestire le sottoscrizioni; rischio di memory leak se non si trascura il ciclo di vita dei componenti.
Compile-time	Aggiornamenti rapidi e precisi; non servono check o notifiche a runtime; adottato da Svelte.	Poca flessibilità; complessita nell'implmentare scenari dinamici non prevedibili a build-time.

In sintesi, i meccanismi di osservazione rappresentano il fulcro della reattività nelle applicazioni web e dunque una variabile fondamentale che influenza il data binding nei vari framework. Al di là delle differenze tecniche tra i vari approcci, tutti si confrontano con problematiche comuni quali efficienza, coerenza e scalabilità, oggetto del dibattito contemporaneo sul binding.

2.4 Problematiche moderne del data binding

Si è tornato a parlare di data binding negli ultimi tempi poichè, pur rappresentando uno strumento indispensabile per costruire interfacce interattive e usabili in app, non è privo di criticità. I framework moderni sono alla costante ricerca di soluzioni innovative, dimostrazione di come la comunità degli sviluppatori stiano ancora cercando un compromesso ottimale tra ergonomia, prevedibilità e performance. In

questa sezione verranno analizzate le principali problematiche che rendono il binding una questione aperta e dibattuta.

2.4.1 Efficienza

Ogni strategia di binding vista sinora presenta costi computazionali in termini di tempo e memoria: il *dirty checking* scala in modo lineare rispetto al numero di watchers; il *Virtual DOM diffing* ha un costo di riconciliazione tra le due versioni che può diventare significativo in scenari complessi e con sottoalberi grandi; la reattività basata su signals o Proxy riduce il numero di aggiornamenti a livello di singoli nodi, ma richiede più accortezza e sofisticazione nella gestione delle dipendenze. Ogni approccio può essere utile in certi scenari, ma molto inefficiente in altri e questo porta alla necessità di trovare compromessi per permettere l'integrazione contemporanea di più tecniche all'interno delle app.

2.4.2 Correttezza e coerenza

Per avere coerenza tra view e model è imperativo garantire sempre la corrispondenza 1:1 tra lo stato mostrato all'utente e i dati sottostanti. Problemi al raggiungimento di questo obiettivo si possono porre soprattutto nel *two-way binding*, in cui è possibile incorrere in cicli infiniti (A aggiorna B, che aggiorna A), tearing (la view mostra uno stato aggiornato solo parzialmente) o anche race condition in presenza di interazioni asincrone. L'interfaccia può risultare molto inaffidabile a causa di aggiornamenti non atomici, dunque assicurarsi che essa rappresenti il più fedelmente possibile ciò che si cela nel model, esattamente quando esso viene cambiato, è una sfida non da poco.

2.4.3 Predicibilità e debuggabilità

È fondamentale per lo sviluppo e la manutenzione del codice che sia semplice ispezionare il flusso di dati. Il binding unidirezionale, implementato ad esempio da React, è una scelta ottimale in quanto favorisce la tracciabilità e la debuggabilità del codice basando il tutto su una singola fonte di verità. Al contrario, i binding impliciti o bidirezionali moltiplicano i punti in cui lo stato può cambiare, presentando complicazioni nei flussi tra componenti e rendendo più complesso il debugging.

2.4.4 Scalabilità architetturale

Applicazioni con dimensioni sempre maggiori tendono ad avere un grafo delle dipendenze sempre più articolato, cosa che, senza accorgimenti, può portare ad un numero quadratico di aggiornamenti. Per questo i framework moderni introducono tecniche di ottimizzazione come la normalizzazione dello stato, la memoization, come ad esempio useMemo() di React [Met25c], o i *pure components*.

2.4.5 Interazione con input complessi

La gestione di form e campi di input richiede particolare attenzione. Il binding non è solo questione di performance, ma anche di user experience: se è troppo aggressivo, i componenti di input si aggiornano ad ogni inserimento di un carattere e questo può causare rallentamenti; se non lo è abbastanza può dare un senso di ritardo nella sincronizzazione. Per far fronte a questo problema i framework moderni adottano un misto tra componenti controllati (valore aggiornato ad ogni inserimento) e non controllati (valore aggiornato su certi eventi) in modo da bilanciare reattività e costi di computazione del binding (per esempio i form di React).

2.4.6 SSR, hydration e streaming

Con la crescente diffusione del *server-side rendering* (SSR) in molte app, il binding deve far fronte anche alla fase di hydration, cioè il riattacco degli event listener agli elementi HTML creati dal server per la pagina e la sincronizzazione tra markup e modello generato dopo il SSR. Soluzioni moderne a questo problema sono ad esempio l'hydration parziale, in cui si fa subito idratazione solo delle parti che servono immediatamente, oppure basata su isole (architettura di Qwik) , in cui si divide la pagina in isole interattive che vengono idratate quando necessario. [Bui23].

2.4.7 Sicurezza

Infine, il data binding può portare problemi di sicurezza se non gestito correttamente. Interpolazioni dirette di input utente nel DOM, senza adeguata sanificazione, espongono infatti al rischio di *cross-site scripting* (XSS), una vulnerabilità molto

comune nelle applicazioni web. I framework moderni tentano di mitigare questo problema applicando strategie di escaping automatico dei contenuti [Vue25j] o fornendo API dedicate, in modo da manifestare esplicitamente l'inclusione di HTML non filtrato.

In conclusione, possiamo affermare che il meccanismo del binding, apparentemente semplice, costituisca uno dei problemi più discussi dell'ultimo decennio, poiché da esso dipendono **l'efficienza** (in termini di tempo di esecuzione e memoria), la **correttezza** dell'applicazione (assenza di stati incoerenti) e, soprattutto, la **qualità dell'esperienza utente**, tutti requisiti fondamentali per sviluppare applicazioni soddisfacenti per il grande pubblico. È per tal motivo che vale la pena analizzare nel dettaglio le modalità con cui i framework più popolari aspirano a portare a termine questa missione, mettendo in evidenza peculiarità, similitudini e differenze tra le diverse implementazioni. Nei prossimi capitoli, dunque, analizzeremo più da vicino come framework quali Angular, React, Vue e Svelte hanno interpretato queste sfide, confrontandone approcci, vantaggi e limiti.

Capitolo 3

Il data binding nei principali framework JavaScript/TypeScript

Come anticipato nella sottosezione 2.4 in questo capitolo analizzeremo come le principali problematiche legate al data binding vengano affrontate nei framework più diffusi, con l'obiettivo di comparare approcci all'implementazione differenti sulla base dei criteri elencati precedentemente. La trattazione sarà strutturata come segue:

- **Angular**, che combina meccanismi classici di binding nei template con strategie moderne di reattività basate sui *Signals*;
- **React**, incentrato sul flusso unidirezionale dei dati e sull'uso del *Virtual DOM* per la riconciliazione;
- **Vue**, che adotta un approccio ibrido, unendo la reattività fine-grained via Proxy con un renderer a Virtual DOM ottimizzato;
- **Svelte**, che elimina il Virtual DOM e sposta la logica di binding a compile-time, riducendo l'overhead a runtime.

In seguito all'analisi dettagliata di ogni tecnologia, una sezione di confronto tabellare evidenzierà i punti comuni e le divergenze, mostrando come ciascun framework abbia interpretato le sfide del binding in modo diverso. Questo discorso comparativo ci permetterà di comprendere meglio i trade-off tra ergonomia, prevedibilità e prestazioni, ponendo in prospettiva le tendenze evolutive del panorama odierno.

3.1 Angular

Nella trattazione seguente si esaminano i principali aspetti del binding in Angular, evidenziando sintassi, meccanismi interni e scelte progettuali.

3.1.1 Da AngularJS ad Angular

Angular nacque originariamente nel 2010 come **AngularJS**, framework basato sul paradigma *Model-View-Controller* (MVC) e caratterizzato da binding bidirezionale implementato tramite il dirty checking. Come menzionato in precedenza, il meccanismo consentiva di propagare automaticamente ogni variazione dal modello alla vista e viceversa, riducendo la quantità di codice da scrivere ma introducendo notevoli problemi di performance nelle applicazioni di grandi dimensioni; dunque, questa soluzione si rivelò poco scalabile.

A partire dal 2016 Google rilascia **Angular 2** (poi noto semplicemente come **Angular**), riscrivendo completamente il framework e abbandonando il vecchio approccio MVC a favore di un'architettura a componenti. In questa nuova versione, il binding diventa prevalentemente monodirezionale ed il ciclo di rilevamento dei cambiamenti è ottimizzato, ma mantiene comunque il supporto al *two-way binding* attraverso la sintassi `[(ngModel)]`, limitandone però l'uso a scenari specifici (soprattutto nei form). Inoltre, Angular moderno non adotta il dirty checking storico di AngularJS; la change detection è orchestrata da Zone.js (o strategie equivalenti) e può essere resa più selettiva con OnPush e Signals, come vedremo più avanti.

3.1.2 Binding nel template

Il termine "template" indica il codice HTML che definisce come viene resa la vista di un componente, dunque il suo DOM. Per fare binding tra model e view, Angular fornisce una vasta scelta di meccanismi: le **interpolazioni**, che consentono di inserire i valori esposti dal componente direttamente nel markup; il **property binding**, tramite il quale è possibile impostare dinamicamente le proprietà di elementi DOM o componenti figli; l'**event binding**, per ascoltare eventi sugli input della view; la combinazione degli ultimi due per ottenere il **two-way binding** mediante la

sintassi `[(ngModel)]` [Ang25d]. L'uso di `ngModel` richiede l'importazione del modulo appropriato (FormsModule) e, dato che è il modo più diretto per implementare il two-way binding, viene maggiormente impiegato per gestire la coerenza dei dati nei form [Ang25a].

```
<!-- Ricevuta fiscale Component -->
<label>Prezzo unitario in euro:
  <input type="number" [(ngModel)]="price">
</label>

<label>Quantita acquistata:
  <input type="number" [(ngModel)]="quantity">
</label>

<label>Percentuale IVA:
  <input type="number" [(ngModel)]="iva">
</label>

<p>Imponibile: {{ price * quantity }} </p>
<p>Prezzo totale: {{ (price * quantity) * (1 + iva/100) }} </p>
```

Figura 3.1: Angular - Two-Way Binding con ngModel

Esempio 2.1.1 (two-way binding con `ngModel`). In questo esempio, che assumiamo si trovi dentro ad un componente già definito, vediamo l'implementazione di una semplicissima ricevuta fiscale che mostra il prezzo totale dei prodotti acquistati e il totale calcolato con IVA inclusa. Come si può notare, i 3 campi di input sono dichiarati con una sintassi intuitiva e minimale che rappresenta tutto ciò che serve per implementare un two-way binding in Angular tramite ngModel, il quale fa direttamente in modo che vista e modello siano sempre aggiornati bidirezionalmente sia in quegli input sia negli elementi HTML nei quali sono stati interpolati, il tutto senza bisogno di event-listeners.

3.1.3 Signals e reattività

Negli ultimi due anni Angular ha introdotto il nuovissimo concetto di *Signals*, ovvero primitive reattive che incapsulano un valore e lo rendono osservabile, consentendo ad Angular di sapere esattamente quando esso cambia e di potervi costruire

computed ed *effect* che rispondano soltanto alle dipendenze effettive. Per *computed* si intende un'espressione derivata basata su uno o più segnali di cui Angular ricalcola automaticamente il valore ogni volta che le dipendenze cambiano, mentre per *effect* intendiamo una funzione che dichiara effetti collaterali da eseguire in risposta ai cambiamenti di uno o più segnali o computed e che viene rieseguita, anch'essa, ad ogni aggiornamento delle dipendenze. Questo approccio permette aggiornamenti mirati delle porzioni di UI che consumano quei segnali, riducendo il lavoro di change detection globale [Ang25b].

```
import { signal, computed, effect } from '@angular/core';

// Signal (variabili reattive)
const prezzoUnitario = signal(10); // in euro
const quantita = signal(2); // numero di articoli

// Computed: totale calcolato automaticamente
const totale = computed(() => prezzoUnitario() * quantita());

// Effetto collaterale
effect(() => {
  console.log(`Prezzo unitario: ${prezzoUnitario()}, Quantita: ${quantita()}, Totale: ${totale()}`);
});

// Aggiornamento di una delle variabili di stato
quantita.set(3); // effettua la notifica e fa rieseguire gli effect/computed necessari
```

Figura 3.2: Angular - Signals, computed, effect

Esempio 2.1.2 (signals, computed, effect). In pratica, le `effect`, tenendo traccia dinamicamente delle `signal` lette, vengono rieseguite automaticamente ad ogni loro cambiamento; inoltre i signal possono essere usati direttamente nei template HTML senza passaggi extra, rendendo la view "consapevole" del signal dal quale dipende. In aggiunta, Angular ha sviluppato delle primitive per rendere reattivi, tramite segnali, anche dati asincroni grazie a funzioni di supporto (es. toSignal() e toObservable()).

3.1.4 Change detection: `Default` vs `OnPush`

Avendo visto come Angular implementa i binding, c'è da capire ora come decida quando fare *change detection*, ovvero aggiornare la vista. Esistono due strategie principali [Ang25c]:

- **Default** — comportamento standard in cui ogni volta che avviene un evento vengono ricontrollati e aggiornati tutti i componenti. È sicuro, ma chiaramente inefficiente su una gran quantità di componenti.
- **OnPush** — strategia più ottimizzata in cui la verifica avviene nel sottoalbero di un componente con `OnPush` solo quando riceve nuovi input via template binding o quando si verifica un evento interno al suo ambito permettendo di saltare interi rami del DOM e risparmiare lavoro di rendering.

In casi particolari è possibile forzare un controllo chiamando `changeDetectorRef.markForCheck()` utile se lo stato viene modificato senza essere intercettato automaticamente dalla strategia OnPush.

```
@Component({
  selector: 'lista-prodotti',
  //template inline
  template: `
    <h3>Lista dei prodotti</h3>
    <ul>
      <li *ngFor="let prodotto of prodotti">{{ prodotto }}</li>
    </ul>
    <button (click)="aggiungiProdotto()">Aggiungi prodotto</button>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush, //tecnica onPush sul componente
})
export class ListaProdottiComponent {
  prodotti = ['Mouse', 'Tastiera'];

  constructor(private cdr: ChangeDetectorRef) {}

  aggiungiProdotto() {
    // simulazione di un aggiornamento non rilevato automaticamente
    setTimeout(() => {
      this.prodotti.push('Monitor');
      this.cdr.markForCheck(); //ricontrollo forzato al prossimo ciclo di
                               //change detection
    }, 1000);
  }
}
```

Figura 3.3: Angular - esempio d'uso di `OnPush` e `ChangeDetectorRef`

Esempio 2.1.3 (component con OnPush).

3.1.5 Form e input

Per quanto riguarda i form, Angular supporta due principali modelli [Ang25a]:

- **Template-driven forms**: form definito nel template HTML usando direttive come `ngModel`, viene aggiornato automaticamente e richiede pochissimo codice, risultando adatto per form di piccole dimensioni;
- **Reactive forms**: form definito principalmente nel codice TypeScript creando a mano l'albero dei controlli tramite `FormControl`, `FormGroup` e validatori; è un modello esplicito a cui il template si lega che fornisce componibilità e validazioni avanzate pur essendo più verboso. È lo standard per form di medie e grandi dimensioni.

I reactive forms si prestano meglio a scenari complessi (validazioni avanzate, form dinamici, testabilità), mentre i template-driven sono più rapidi da scrivere per casi semplici.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({ //decoratore TypeScript
  selector: 'dati-personali-utente',
  templateUrl: './dati-utente.component.html'
})
export class FormDatiUtenteComponent {
  formProfiloUtente = new FormGroup({
    nome: new FormControl('', { nonNullable: true, validators:
    [Validators.required] }),
    email: new FormControl('', { nonNullable: true, validators:
    [Validators.required, Validators.email] }),
    eta: new FormControl(21, { nonNullable: true, validators:
    [Validators.min(0)] })
  });

  submit() {
    if (this.formProfiloUtente.valid) {
      console.log('Dati che stai inviando:', this.formProfiloUtente.getRawValue());
    }
  }
}
```

```
//Template file
<form [formGroup]="formProfiloUtente" (ngSubmit)="submit()">
  <h3>Inserisci i tuoi dati personali</h3>

  <label>Nome:
    <input type="text" formControlName="nome" />
  </label>

  <label>Email:
    <input type="email" formControlName="email" />
  </label>

  <label>Eta:
    <input type="number" formControlName="eta" />
  </label>

  <p>Anteprima dei dati inseriti: {{ formProfiloUtente.value | json }}</p>

  <button type="submit" [disabled]="formProfiloUtente.invalid">Invia</button>
</form>
```

Figura 3.4: Angular - Reactive Form per l'inserimento di dati utente

Esempio 2.1.4 (Reactive Form). Dall'esempio, in cui presentiamo un form da compilare con alcuni dati personali dell'utente, si intuisce che il model viene definito nel codice TypeScript tramite `FormControl` e `FormGroup`, mentre nel template HTML ci si limita a collegare i controlli dichiarati con le direttive `formGroup` e `formControlName`. In questo modo, cambiamenti fatti dall'utente nell'input aggiornano automaticamente il modello e, viceversa, eventuali modifiche fatte al modello a livello di codice vengono propagate immediatamente alla vista.

3.1.6 SSR e hydration

Angular supporta il rendering lato server tramite *Angular Universal*, una soluzione che permette il SSR, dunque il server genera in anticipo l'HTML, cosicchè la pagina sia visibile più velocemente e lo manda al client per l'hydration, ovvero il riallacciamento dei listener agli eventi e la ristabilizzazione del binding. Vantaggio di questo approccio è quello di aumentare la velocità di caricamento della pagina e di non lasciare lo user ad attendere che il client carichi tutto insieme da capo.

3.1.7 Considerazioni e trade-off

- **Flessibilità**: Angular mette a disposizione diversi modi per fare binding e gestire gli stati: binding del template (interpolazioni, `ngModel`) per form semplici; reactive forms per form complessi e Signals per reattività fine-grained. Ciò permette di scegliere lo strumento più adatto al caso d'uso ma aumenta la superficie concettuale che il team di sviluppo deve padroneggiare.

- **Reattività**: Angular, attraverso i `Signals`, supporta un modello di reattività *fine-grained*, ovvero un modello in cui il framework provvede ad aggiornare soltanto le parti della vista effettivamente dipendenti dai dati cambiati, riducendo gli aggiornamenti superflui al minimo. Infatti, è in grado di identificare esattamente quali componenti sono da aggiornare e modificare solo quelli. Ciò riduce il numero di ricalcoli rispetto al modello tradizionale, ma richiede maggiore disciplina nella gestione delle dipendenze e delle sottoscrizioni.

- **Efficienza**: la strategia `OnPush` e l'uso di Signals riducono notevolmente il lavoro di verifica e aggiornamento rispetto al modello *check always*, ma richiedono particolare attenzione nella gestione degli stati.
- **Tooling e maturità**: Angular fornisce API formali (ChangeDetectorRef, FormsModule, Angular Universal) e best practice consolidate perfette per team medio/grandi e processi aziendali; tuttavia la curva d'apprendimento è significativa, pur essendo dotato di documentazione robusta ed ergonomica.

In conclusione, Angular è una piattaforma che cerca di conciliare ergonomia e controllo sulla performance, rendendo possibile ottenere aggiornamenti più mirati e prevedibili, sebbene ciò imponga una maggiore attenzione all'architettura delle componenti e alla disciplina nell'uso delle API di stato.

3.2 React

React, sviluppato da Facebook nel 2013, è radicalmente diverso rispetto ad Angular poichè utilizza il **binding unidirezionale** e non fornisce alcun meccanismo automatico di sincronizzazione bidirezionale, ma va tutto sviluppato esplicitamente. In React la vista è definita tramite **JSX**, una sintassi che estende JavaScript permettendo di generare markup simile a HTML; i dati vengono passati dai componenti genitori ai figli tramite `props`, che determinano il comportamento e l'aspetto di questi ultimi, mentre l'utente può aggiornare lo stato interno tramite gli `hook` [Met25e].

3.2.1 Binding e JSX

Il binding in React avviene tramite l'utilizzo di hooks, costrutti speciali che permettono di aggiornare esplicitamente gli stati, come ad esempio `useState`, mentre gli eventi vengono gestiti tramite handler (ad esempio `onChange`) all'interno dei componenti contenenti gli input con cui l'utente interagisce nella view [Met25d].

```jsx
import { useState } from "react";

function RicevutaComponent() {
  const [prezzo, setPrezzo] = useState(0);
  const [quantita, setQuantita] = useState(1);
  const [iva, setIva] = useState(22);

  const imponibile = prezzo * quantita;
  const prezzoTotale = imponibile * (1 + iva / 100);

  return (
    <>
      <h3>Ricevuta fiscale</h3>

      <label>Prezzo unitario in euro:
        <input
          type="number"
          value={prezzo}
          onChange={e => setPrezzo(Number(e.target.value))}
        />
      </label>

      <label>Quantita:
        <input
          type="number"
          value={quantita}
          onChange={e => setQuantita(Number(e.target.value))}
        />
      </label>

      <label>Percentuale IVA:
        <input
          type="number"
          value={iva}
          onChange={e => setIva(Number(e.target.value))}
        />
      </label>

      <p>Imponibile: {imponibile} </p>
      <p>Prezzo totale: {prezzoTotale} </p>
    </>
  );
}
```

Figura 3.5: React - Sintassi di binding in input controllati con useState

Esempio 2.2.1 (input controllato con useState) In questo esempio, in cui implementiamo di nuovo una semplice ricevuta, usiamo l'hook useState per definire delle variabili (`prezzo`, `quantita`, `iva`) e dei setter di quelle variabili, che sono l'unico modo con cui si può modificare il valore di esse e fare binding. Il valore dei vari `input` mostrati all'utente sono legati alle suddette variabili di stato, dunque ad ogni cambiamento si scatena l'handler `onChange` che, come esplicitato dalla sintassi, aggiorna gli stati tramite `setPrezzo`, `setQuantita` o `setIva`, causando ogni volta

un re-render del componente.

3.2.2 Virtual DOM e riconciliazione

Per fare change detection e per la riconciliazione, React utilizza un **Virtual DOM**: una rappresentazione in memoria dell'albero del DOM. Ogni volta che uno stato (o le props) cambiano, React calcola la differenza (*diffing*) tra la nuova versione del Virtual DOM e quella precedente, applicando al DOM reale solo le modifiche necessarie. Questa tecnica migliora la prevedibilità e costa notevolmente di meno rispetto a controllare in modo esaustivo il DOM direttamente, poiché lavora solo sulle differenze nei nodi realmente modificati, ma introduce comunque un overhead di calcolo, soprattutto in applicazioni di grandi dimensioni con molti componenti che renderebbero la costruzione del nuovo virtual DOM e del calcolo delle differenze molto pesante. A causa di ciò, la reattività di React è considerata di tipo *coarse-grained*, ovvero segue un modello in cui un cambiamento di stato provoca il ricalcolo o il re-render di intere porzioni (ad es. un componente completo), anche se solo una parte limitata avrebbe realmente bisogno di aggiornamento.

3.2.3 Gestione dello stato

React fornisce vari strumenti per la gestione dello stato:

- `useState`: per mantenere lo stato locale di un componente; provoca il re-rendering del componente;
- `useReducer`: per stati più complessi che hanno più possibili transizioni;
- `useContext`: per condividere lo stato tra più componenti senza doverlo passare manualmente tra le props componente per componente.

Tutte queste tecniche richiedono il re-rendering dei componenti ad ogni modifica sugli stati, cosa che può impattare molto negativamente sulla reattività. Per questo motivo React mette a disposizione degli approcci per mitigare questa potenziale inefficienza.

3.2.4 Ottimizzazioni del rendering

Per ridurre i render inutili, React mette a disposizione tecniche come:

- `React.memo`: evita la ricreazione dei valori durante il re-render di un componente se le props non sono cambiate;
- `useMemo`: memorizza il risultato di funzioni costose in modo da non dover essere continuamente ricalcolate;
- `useCallback`: memorizza le funzioni per prevenire ricreazioni superflue ad ogni chiamata.

```
import { useMemo } from "react";

function ExampleComponent({ items }) {
//somma dei valori in items
  const sommaTotale = useMemo(() => items.reduce((a, b) => a + b, 0), [items]);
  return <p>Somma totale: {sommaTotale}</p>;
}
```

Figura 3.6: React - Esempio di utilizzo di useMemo

Esempio 2.2.2 (uso di useMemo). In questo esempio vediamo una somma di numeri che senza `useMemo` verrebbe ricalcolata ad ogni re-render del componente anche se items non viene modificato. Utilizzando `useMemo`, invece, salviamo il valore di `sommaTotale` attraverso i render finchè non c'è un'effettiva modifica su items, ovvero la sua dipendenza.

Nonostante la validità e l'utilità dell'uso di questi hook per ottimizzare le prestazioni durante i rendering, nel 2025 è stato introdotto il nuovissimo **React Compiler**, che si occupa a compile-time di praticare memoization automaticamente senza bisogno di ulteriore codice di ottimizzazione basato su `useMemo` e `useCallback`. [Met25b].

3.2.5 Effetti collaterali e riferimenti

Oltre agli strumenti per la gestione dello stato, React fornisce anche hook come `useEffect`, che consente di dichiarare *side effects* (ad esempio fetch di dati, sotto-

scrizioni a eventi o aggiornamenti del DOM) eseguiti in risposta ai cambiamenti dello stato o delle `props`; e `useRef`, che permette di creare un contenitore persistente che mantiene valori mutabili in modo da non causare re-render ad ogni modifica. Questi hook non gestiscono direttamente lo stato visibile dell'applicazione, ma offrono flessibilità e ottimizzazione sul binding sottostante tra stati e componenti [Met25d].

3.2.6 Form e input

React gestisce generalmente i form tramite **controlled components**, in cui ogni input è legato esplicitamente allo stato del componente invece di mantenere semplicemente il suo stato interno. Questo permette ad ogni modifica dell'utente di aggiornare immediatamente la view [Met25a].

```
import { useState } from "react";
function FormDatiUtente() {
  const [datiForm, setDatiForm] = useState({
    nome: "",
    email: "",
    eta: ""
  });
  const handleCambioStato = (e) => {
    const { name, value } = e.target;
    setDatiForm((prevData) => ({
      ...prevData,
      [name]: value
    }));
  };
  const handleInvioForm = (e) => {
    e.preventDefault();
    console.log("Dati che stai inviando:", datiForm);
  };
  return (
    <form onSubmit={handleInvioForm}>
      <h3>Inserisci i tuoi dati personali</h3>
      <label>Nome: <input type="text" name="nome" value={datiForm.nome} onChange={handleCambioStato} />
      </label>
      <label>Email: <input type="email" name="email" value={datiForm.email} onChange={handleCambioStato} />
      </label>
      <label>Eta: <input type="number" name="eta" value={datiForm.eta} onChange={handleCambioStato}/>
      </label>
      <p>Anteprima dati inseriti: {JSON.stringify(datiForm)}</p>
      <button type="submit">Invia</button>
    </form>
  );
}
```

Figura 3.7: React - Controlled Form per l'inserimento di dati utente

Esempio 2.2.3 (React form). Dall'esempio sopra esposto, in cui proponiamo un semplice form da compilare con dati personali dell'utente, si vede esattamente come React riesca ad implementare two-way binding all'interno di componenti complessi come i form, usando comandi espliciti: gli stati e i controlled components sono il modo esplicito e più utilizzato di fare in modo che view e model rimangano aggiornati bidirezionalmente pur non avendo meccanismi automatici come Angular.

3.2.7 SSR e hydration

React, di base, è una libreria client-side, dunque gira completamente sul browser, dove genera il DOM e tutti gli aggiornamenti. Tuttavia è ormai da vari anni che viene affiancato a **Next.js**, che estende React e aggiunge server-side rendering dei componenti e degli elementi HTML ai quali, poi, vengono riallacciati gli event handler e gli stati durante la fase di hydration [Ver25]. Next fornisce inoltre API per la *static site generation* (SSG) e lo *streaming*: la prima è una tecnica che pre-genera le pagine HTML a build time, rendendole subito disponibili come file statici e riducendo i tempi di risposta lato server; il secondo è una funzionalità introdotta con React 18 che permette di inviare al client l'HTML in modo incrementale (chunk), mostrando la pagina a pezzi man mano che i dati diventano disponibili.

3.2.8 Considerazioni e trade-off

- **Flessibilità**: React non impone un'architettura rigida ma fornisce primitive minimali (stato, props, hook) da cui è possibile costruire soluzioni personalizzate. Questo approccio garantisce grande espressività e libertà nello scegliere il paradigma migliore in ogni singolo caso, tuttavia richiede attenzione per evitare di produrre soluzioni troppo eterogenee difficili da mantenere su progetti di grandi dimensioni.

- **Reattività**: in React il modello è tipicamente *coarse-grained*, cioè ogni modifica di stato scatena un re-render dell'intero componente, indipendentemente dal numero di nodi effettivamente toccati. Questo approccio semplifica il flusso dei dati e la prevedibilità, ma può introdurre calcoli ridondanti.

- **Efficienza**: l'uso del Virtual DOM è molto utile per ridurre le inconsistenze tra stato e vista e per semplificare il modello mentale per gli sviluppatori, ma ha un costo computazionale elevato in applicazioni complesse. Tuttavia abbiamo visto come possibile soluzione a questo problema il React Compiler, che riduce parte di questo overhead.
- **Tooling e maturità**: l'ecosistema React è vasto e consolidato (Next.js, Redux, DevTools), ben documentato e ampiamente adottato, ma richiede disciplina per mantenere leggibilità e coerenza in progetti complessi. Può risultare un po' impegnativo padroneggiare l'ecosistema completo, ma in generale la curva di apprendimento è relativamente rapida.

In conclusione, React è caratterizzato dalla semplicità delle astrazioni e dalla prevedibilità del flusso dei dati, ottenuti grazie al binding unidirezionale e al modello basato su `props` e `state`. La combinazione tra Virtual DOM, hook e, più recentemente, il React Compiler, consente di ottenere applicazioni scalabili e performanti, pur richiedendo disciplina architetturale. La flessibilità e il grande ecosistema che lo circonda sono i punti di forza di React.

3.3 Vue

Vue, creato da Evan You nel 2014, è un *progressive framework* di JavaScript: è stato, infatti, progettato principalmente per costruire UI interattive e scalabili in maniera incrementale, ovvero può essere usato in partenza come library leggera per arricchire pagine esistenti fino ad arrivare a diventare una piattaforma completa per sviluppare interfacce di qualsiasi complessità. È un framework con architettura *component based* e la sua filosofia mette insieme semplicità dichiarativa e potenza tramite API reattive moderne basate su *Proxy* e Composition API [You25; Wik25b].

3.3.1 Binding e template syntax

La sintassi di binding di Vue nel template è semplice e dichiarativa e presenta varie tecniche per connettere model e view [Vue25h]:

- **Interpolazioni**: per inserire valori di stato nel markup tramite `{{ }}`;
- **Property binding**: serve per collegare valori dinamici alle proprietà di elementi e componenti tramite `v-bind` (o la scorciatoia `:`);
- **Event binding**: con `v-on` (o la scorciatoia `@`) si collegano funzioni agli eventi del DOM;
- **Two-way binding**: la direttiva `v-model` combina `v-bind` e `v-on` per implementare aggiornamenti bidirezionali nei campi di input legati allo stato.

```
<template>
  <h3>Ricevuta fiscale</h3>

  <label>Prezzo unitario in euro:
    <input type="number" v-model.number="prezzo" />
  </label>

  <label>Quantita:
    <input type="number" v-model.number="quantita" />
  </label>

  <label>Percentuale IVA:
    <input type="number" v-model.number="iva" />
  </label>

  <p>Imponibile: {{ (prezzo * quantita) }} </p>
  <p>Prezzo totale: {{ ((prezzo * quantita) * (1 + iva / 100)) }} </p>
</template>

<script setup>
import { ref } from 'vue'

const prezzo = ref(0)
const quantita = ref(1)
const iva = ref(22)
</script>
```

Figura 3.8: Vue - Esempio sintassi binding con v-model

Esempio 2.3.1 (uso di v-model) In questo esempio, in cui vediamo sempre la nostra finta ricevuta fiscale, le variabili reattive `prezzo`, quantità e `iva` sono sincronizzate automaticamente con gli input tramite `v-model`, senza la gestione manuale di eventi 'onChange', implementando un aggiornamento istantaneo bidirezionale. Oltre all'uso di `v-model` vediamo chiaramente l'interpolazione dei dati all'interno dei

paragrafi "Imponibile" e "Prezzo totale" e l'uso di `ref`, che permette di rendere le tre variabili reattive e osservabili e collegarle direttamente ai componenti dell'interfaccia che dipendono da esse, senza bisogno di specificare direttive esplicite [You25].

3.3.2 Reattività con Proxy

A partire dalla versione più recente di Vue, Vue 3, la reattività è basata su Proxy, che intercetta dinamicamente tutte le letture e scritture sui dati, anche se sono innestati, introducendo un modello di reattività *fine-grained*. Grazie a questo approccio, Vue non controlla tutto l'albero del DOM, ma sa esattamente quale parte dell'interfaccia utente (UI) aggiornare in base a quali dati sono stati letti. Le API principali utilizzate sono [Vue25e]:

- `reactive()`: prende un oggetto e lo trasforma in uno reattivo, avvolgendolo con Proxy, in modo che tenga traccia delle proprietà lette/scritte;
- `ref()`: incapsula valori primitivi o oggetti in wrapper reattivi in modo da renderli reattivi e gli fornisce una proprietà .value per accedere al contenuto;
- `computed()`: definisce valori derivati da altri reattivi che si aggiornano automaticamente quando le dipendenze cambiano;
- `watch()`: osserva cambiamenti specifici su un dato e reagisce con callback ogni volta che c'è una modifica. È utile per side effects.

```vue
<script setup>
import { reactive, computed } from 'vue'

const dati = reactive({
  prezzo: 10,
  quantita: 2
})

// valore derivato calcolato automaticamente
const totale = computed(() => dati.prezzo * dati.quantita)
</script>

<template>
  <h3>Ricevuta</h3>

  <label>Prezzo unitario in euro:
    <input type="number" v-model.number="dati.prezzo" />
  </label>

  <label>Quantita:
    <input type="number" v-model.number="dati.quantita" />
  </label>

  <p>Prezzo totale: {{ totale }} </p>
</template>
```

Figura 3.9: Vue - Esempio di reattività con reactive e computed

Esempio 2.3.2 (reattività con reactive e computed). In questo esempio, al primo render Vue valuta le espressioni nel template: legge il valore di `totale`, che a sua volta dipende da `dati.prezzo` e `dati.quantita` e costruisce il DOM iniziale. Alla modifica di uno dei due input, il sistema aggiorna automaticamente le proprietà osservate di `dati`; Vue, quindi, invalida il computed associato e lo ricalcola, aggiornando soltanto il nodo del DOM che utilizza `totale`. In questo modo il sistema, attraverso Proxy, fa sì che solo i nodi che dipendono da `dati.prezzo`, `dati.quantita` o `totale` vengano aggiornati, ottenendo un aggiornamento mirato ed efficiente senza bisogno di diffing sull'intero albero del DOM [Vue25b].

3.3.3 Virtual DOM e riconciliazione

Anche Vue, come React, mantiene in memoria una rappresentazione dell'albero del DOM. Quando lo stato cambia , Vue costruisce un nuovo Virtual DOM col quale fare diffing per capire quali nodi aggiornare. Tuttavia, con l'avvento di Vue 3, è stato introdotto il template compiler, che gestisce in maniera diversa nodi statici e

nodi dinamici: i primi vengono estratti e memorizzati una sola volta cosi da non doverli ricontrollare ad ogni render; gli altri vengono accoppiati con delle etichette che permettono a Vue di sapere esattamente quale parte del nodo è dinamica e deve essere aggiornata, senza fare diffing sull'intero albero. [Vue25f]. Queste ottimizzazioni permettono un approccio misto tra diff e push e di mitigare l'overhead tipico del Virtual DOM, avvicinandosi ad una reattività fine grained, ma godendo contemporaneamente dei vantaggi di entrambi.

3.3.4 Gestione dello stato

Vue supporta vari livelli di gestione dello stato:

- Stato locale con `ref` / `reactive`: lo stato si trova all'interno del componente e viene automaticamente tracciato da Vue per aggiornare la UI collegata ad esso [Vue25e];
- Stato condiviso via Composition API: la Composition API è un insieme di funzioni introdotte in Vue 3 (`setup()`, `ref()`, `reactive()`, `computed()`, ecc.) che consente di definire la logica dei componenti in maniera dichiarativa e modulare, favorendo il riuso di codice rispetto alla precedente Options API. Grazie a queste nuove introduzioni, per condividere uno stato da un componente padre a uno o più figli, invece di passarlo tramite props per tanti livelli si usano le funzioni `provide/inject`. Provide() permette al componente padre di "fornire" un valore, e inject() ai figli di "iniettarlo" direttamente [Vue25i];
- Stato globale con state manager: con l'avvento di Vue 3 si è diffuso l'utilizzo di **Pinia**, uno strumento che gestisce lo stato globale, accessibile a tutti i componenti, grazie ad uno store centralizzato completamente integrato con la Composition API. Questa soluzione è leggera, modulare e in piena compatibilità con la reattività di Vue, perciò permette di semplificare la definizione di store globali e il loro utilizzo nei componenti [Mor25].

3.3.5 Form e input

In Vue i form sono gestiti grazie alla direttiva `v-model`, che permette il two-way binding dei campi coinvolti e che abbiamo osservato nell'esempio 2.3.1 [Vue25c]. Oltre a ciò, vengono forniti anche modificatori per controllare il comportamento del binding [Vue25d]:

- `.lazy`: aggiorna lo stato solo sull'evento 'change', invece di farlo ad ogni carattere digitato nell'input. In questo modo riduce il numero di update e aumenta la performance;
- `.number`: forza la conversione del valore in numero;
- `.trim`: rimuove automaticamente spazi superflui dalle stringhe.

3.3.6 SSR e hydration

Vue supporta il rendering lato server tramite le API ufficiali di **SSR**, consentendo di generare l'HTML sul server e poi collegarlo al client tramite **hydration** [Vue25g]. Tuttavia, nella pratica viene spesso usato **Nuxt.js**, l'equivalente di Next.js per Vue, che estende queste funzionalità aggiungendo anche **SSG** (Static Site Generation) e modalità ibride come lo **streaming** o la **partial hydration** (una tecnica che consiste nell'idratare solamente le parti interattive della pagina, i "componenti interattivi", lasciando statiche le parti non dinamiche in modo da ridurre il carico del client), riducendo l'overhead a runtime [Nux25].

3.3.7 Considerazioni e trade-off

- **Flessibilità**: Vue adotta l'approccio *progressive framework*, ovvero consente di partire dalla sintassi semplice e dichiarativa dei template per poi incrementare con Composition API quando serve complessità. Questa flessibilità lo rende adatto sia a progetti piccoli che ad applicazioni enterprise, anche se bisogna dichiarare bene le convenzioni tra i componenti del team per evitare frammentazione di stili.

- **Reattività**: il modello adottato da Vue è quasi completamente *fine-grained* ed è basato su Proxy. Questo approccio è più efficiente rispetto al modello coarse-grained, ma richiede attenzione quando si gestiscono stati complessi, soprattutto nell'uso dei watcher e delle computed properties.
- **Efficienza**: nonostante utilizzi il Virtual DOM, le ottimizzazioni compile-time (hoisting statico, patch flags) alleggeriscono il Virtual DOM e migliorano le performance runtime. Questo lo rende competitivo con framework senza Virtual DOM (come Svelte), pur mantenendo la flessibilità di un renderer generico.
- **Tooling e maturità**: l'ecosistema Vue è maturo (CLI, Vite, Nuxt, Pinia), la documentazione è chiara e dettagliata e la curva di apprendimento accessibile: Vue viene considerato molto user friendly e intuitivo grazie alla sua sintassi semplice e basata sui template HTML tradizionali [Sof24].

In conclusione, Vue è popolare per il bilanciamento interessante che offre tra la semplicità tipica delle librerie leggere e la potenza delle strutture avanzate. La reattività fine-grained e le ottimizzazioni interne lo configurano come una valida alternativa ai modelli di binding tradizionali, con un'eccellente esperienza di sviluppo e performance competitive.

3.4 Svelte

Svelte, creato da Rich Harris nel 2016, si distingue dagli altri framework per la sua natura di **compiler**: traduce i componenti in JavaScript ottimizzato che aggiorna direttamente il DOM al posto di generare un runtime pesante. Ciò rende superfluo l'uso del Virtual DOM e riduce drasticamente l'overhead a runtime [Sve25a; Kle23].

3.4.1 Binding e sintassi

La sintassi adottata da Svelte per il binding è dichiarativa ed è molto semplice: si usa `{}` per le interpolazioni, la direttiva `bind:` per il binding bidirezionale e `on:` per la gestione degli eventi. Questo approccio consente di legare variabili e input senza codice aggiuntivo [Sve25e].

```
<script>
  let prezzo = 0;
  let quantita = 1;
  let iva = 22;

  $: imponibile = prezzo * quantita;
  $: prezzoTotale = imponibile * (1 + iva / 100);
</script>

<h3>Ricevuta fiscale</h3>

<label>Prezzo unitario in euro:
  <input type="number" bind:value={prezzo} />
</label>

<label>Quantita:
  <input type="number" bind:value={quantita} />
</label>

<label>Percentuale IVA:
  <input type="number" bind:value={iva} />
</label>

<p>Imponibile: {imponibile} </p>
<p>Prezzo totale: {prezzoTotale} </p>
```

Figura 3.10: Svelte - Sintassi di binding

Esempio 2.4.1 In questo esempio di una ricevuta fiscale vediamo l'utilizzo della direttiva `bind:` e l'uso di `$:`, approfondito nella sezione successiva, che rendono reattive le variabili e consentono l'aggiornamento bidirezionale nei tre input e negli elementi che dipendono da essi, ovvero `imponibile` e `prezzoTotale`.

3.4.2 Reattività a compile-time

In Svelte la reattività non è gestita a runtime, bensì dal compilatore attraverso l'operatore `$:` (detto anche *reactive statement*), che segnala automaticamente a Svelte le variabili derivate da ricalcolare quando cambiano le dipendenze. Non si usano né dirty checking né Proxy: gli aggiornamenti sono mirati e determinati già in fase di build grazie alle istruzioni JavaScript generate appositamente per aggiornare solo le parti della UI coinvolte [Ric21].

3.4.3 Gestione dello stato

Svelte gestisce gli stati locali ai componenti con variabili che, grazie al compilatore, diventano automaticamente reattive. Gli stati condivisi tra più componenti, invece, sono gestiti tramite le primitive di **store** (`writable`, `readable`, `derived`) che propagano direttamente i cambiamenti e sostituiscono la necessità di librerie esterne come Redux o Vuex [Sve25b]. Queste primitive sono astrazioni reattive integrate nel framework che rappresentano contenitori di stato condivisibile e offrono un'interfaccia standard per leggere i valori e ricevere aggiornamenti in tempo reale. Nei template è possibile accedere direttamente al valore di uno store tramite la sintassi `$store`.

3.4.4 Assenza di Virtual DOM

Come anticipato sopra, a differenza degli altri framework, Svelte non conserva una versione virtuale del DOM: il compilatore analizza il codice dell'app a compile-time e genera direttamente del codice Javascript che si occupa di aggiornare solo i nodi interessati senza bisogno di alcun diffing. Questo meccanismo è la grande novità che ha posto Svelte sotto i riflettori, poiché riduce drasticamente il costo computazionale a runtime e garantisce performance spesso superiori grazie all'eliminazione di un livello di astrazione [Kra24].

3.4.5 Form e input

Grazie alla direttiva `bind:value` la gestione dei form in Svelte è molto leggera e facilitata. Possiamo fare riferimento all'esempio 2.4.1 per capire come vengano implementati i form classici con questo framework che implementa il two-way binding con sintassi diretta e senza bisogno di gestione di eventi onChange, onInput e così via.

3.4.6 SSR e hydration

Anche Svelte, come i framework visti finora, si appoggia su un framework chiamato **SvelteKit** per supportare il rendering lato server, un equivalente di Next in React e di Nuxt per Vue [Sve25d]. SvelteKit abilita **SSR**, **SSG**, ed altre modalità come lo

streaming [Sve25c]. L'‘hydration‘ collega gli eventi client-side al markup generato dal server.

3.4.7 Considerazioni e trade-off

- **Flessibilità**: Come si può dedurre dalle informazioni raccolte finora, Svelte privilegia semplicità e immediatezza: meno codice da scrivere significa meno concetti da padroneggiare. Questo aspetto lo rende ideale per prototipi e progetti rapidi di piccole/medie dimensioni; per progetti enterprise, invece, può portare a mancanza di coerenza architetturale a causa della mancanza di convenzioni forti.
- **Reattività**: anche il modello di Svelte è *fine-grained*, poiché vengono aggiornati soltanto i nodi dipendenti dalle variabili modificate.
- **Efficienza**: Grazie all'assenza di Virtual DOM e all'uso del compilatore, Svelte riduce notevolmente l'overhead e i tempi di esecuzione, presentando una performance molto vicina a quelle "native" del browser [Kra24].
- **Tooling e maturità**: Grazie a SvelteKit, Svelte è cresciuto, ha ampliato i suoi usi e risulta ancora in crescita. Possiede uno dei sistemi di binding più semplici, intuitivi e veloci tra tutti i framework, migliora di molto l'esperienza sia dal lato user che dal lato programmer e la curva di apprendimento è veloce, tuttavia l'ecosistema resta ancora più piccolo e meno consolidato rispetto, ad esempio, a React o Angular [Kra24].

In conclusione, Svelte fornisce l'approccio più innovativo al binding, grazie allo spostamento della logica a compile-time, ottenendo aggiornamenti precisi con un modello semplice da usare, ma con un ecosistema ancora in evoluzione.

3.5 Analisi comparativa tra i framework

Alla luce delle analisi dettagliate condotte finora, in questa sezione giungiamo al confronto finale tra i quattro principali framework: si evidenziano differenze e punti

in comune riguardo uso del binding, reattività, efficienza, espressività, semplicità di riutilizzo e di apprendimento del codice [Tec23; Sch24; Dha24; Sof24; Bui24; Har18; Pet24]. Si propone innanzitutto uno specchietto comparativo riassuntivo per mettere in luce le caratteristiche più significative delle quattro tecnologie in modo visivo e intuitivo.

3.5.1 Matrice comparativa

Criterio	Angular	React	Vue	Svelte
Implementazione binding	Interpolazione, property e event binding, ngModel	JSX, hooks, handler espliciti	Interpolazione, v-bind, v-on, v-model	{}, bind:, on:
Tipo di binding principale	Two-way (ngModel)	One-way (two-way via handler indiretti)	Two-way (v-model)	Two-way (bind:)
Modello di reattività	Fine-grained	Coarse-grained	Fine-grained	Fine-grained
Rendering/ Change detection	Zone.js	Virtual DOM diffing	Virtual DOM + patch flags	Aggiornamenti diretti tramite compiler
SSR e hydration	Angular Universal	Next.js	Nuxt.js	SvelteKit
Ergonomia	Robusto e completo, ma complesso, curva di apprendimento ripida	Flessibile e minimale, ma più verboso	Sintassi progressiva, documentazione chiara, beginner-friendly	Sintassi semplice ed immediata, curva di apprendimento veloce
Criticità tipiche	Complessità concettuale e boilerplate	Memoization manuale, rischio di re-render ridondanti	Rischio di complessità in stati annidati	Ecosistema giovane e con meno convenzioni forti

Tabella 3.1: Confronto riassuntivo tra i principali framework sul data binding.

3.5.2 Analisi sintattica

Angular

```
<label>Prezzo unitario in euro: <input type="number" [(ngModel)]="price">
</label>
<label>Quantita acquistata: <input type="number" [(ngModel)]="quantity">
</label>
<label>Percentuale IVA: <input type="number" [(ngModel)]="iva">
</label>
<p>Imponibile: {{ price * quantity }} </p>
<p>Prezzo totale: {{ (price * quantity) * (1 + iva/100) }}</p>
```

React

```
function RicevutaComponent() {
  const [prezzo, setPrezzo] = useState(0); const [quantita, setQuantita] = useState(1); const [iva, setIva] = useState
        (22);
  const imponibile = prezzo * quantita; const prezzoTotale = imponibile * (1 + iva / 100);
  return ( <> <h3>Ricevuta fiscale</h3>
      <label>Prezzo unitario in euro: <input type="number" value={prezzo} onChange={e => setPrezzo(Number(e.target.value)
           )} />
      </label>
      <label>Quantita: <input type="number" value={quantita} onChange={e => setQuantita(Number(e.target.value))}/>
      </label>
      <label>Percentuale IVA: <input type="number" value={iva} onChange={e => setIva(Number(e.target.value))} />
      </label>
      <p>Imponibile: {imponibile} </p>
      <p>Prezzo totale: {prezzoTotale} </p> </> );
}
```

Vue

```
<template>
  <h3>Ricevuta fiscale</h3>
  <label>Prezzo unitario in euro: <input type="number" v-model.number="prezzo"/></label>
  <label>Quantita: <input type="number" v-model.number="quantita" /></label>
<label>Percentuale IVA: <input type="number" v-model.number="iva" /></label>
  <p>Imponibile: {{ (prezzo * quantita) }} </p>
  <p>Prezzo totale: {{ ((prezzo * quantita) * (1 + iva / 100)) }} </p>
</template>

<script setup>
import { ref } from 'vue'
const prezzo = ref(0) const quantita = ref(1) const iva = ref(22)
</script>
```

Svelte

```
<script>
  let prezzo = 0; let quantita = 1; let iva = 22;
  $: imponibile = prezzo * quantita;
  $: prezzoTotale = imponibile * (1 + iva / 100);
</script>
<h3>Ricevuta fiscale</h3>
<label>Prezzo unitario in euro: <input type="number" bind:value={prezzo} /></label>
<label>Quantita:<input type="number" bind:value={quantita} /></label>
<label>Percentuale IVA:<input type="number" bind:value={iva} /></label>
<p>Imponibile: {imponibile} </p>
<p>Prezzo totale: {prezzoTotale} </p>
```

Figura 3.11: Specchietto comparativo sulla sintassi di binding

Dal punto di vista sintattico, come si può desumere dalla tabella 2.1 e dalla Figura 3.11, Angular è il più verboso e strutturato: ogni componente richiede dichiarazioni specifiche e ripetitive che garantiscono robustezza, ma aumentano il boilerplate. React, invece, è molto più esplicito: il binding manuale garantisce massima flessibilità, ma produce anche una maggiore verbosità logica [Bui24; Pet24]. Vue utilizza un approccio intermedio: la sintassi è semplice e dichiarativa, ma può ricadere in complessità superflue a causa della Composition API [Sof24]. Svelte è il più conciso e naturale tra tutti: è quasi del tutto privo di sintassi extra e fornisce modi intuitivi di collegare l'interfaccia e le variabili [Har18; Bui24].

Dal punto di vista della curva di apprendimento, Vue e Svelte risultano sicuramente più beginner-friendly grazie alla sintassi progressiva e intuitiva e all'assenza di concetti troppo astratti. Angular e React richiedono tempi più lunghi per essere padroneggiati, specialmente in progetti più corposi [Sof24].

3.5.3 Efficienza e performance

Sul piano prestazionale, tutti i framework sono più o meno equivalenti, con alcune piccole differenze da tenere in considerazione in base alle esigenze. Il framework più efficiente è sicuramente Svelte, grazie all'eliminazione del Virtual DOM e al suo innovativo approccio compile-time [Kra24]. Altro framework che riesce ad ottenere performance ad alto livello è Vue 3, grazie alle ottimizzazioni messe in atto per ottimizzare l'uso del Virtual DOM. React, classicamente, paga un costo significativo al diffing del Virtual DOM [Sof24]; tuttavia, dal 2025 ha introdotto il React Compiler che ha avvicinato molto le sue prestazioni a quelle di Svelte automatizzando la memoization e riducendo molto il lavoro del programmatore [Ras25]. Angular rimane il più pesante (anche se non in maniera davvero significativa), specialmente in progetti di grandi dimensioni, ma l'introduzione dei Signals ha portato molta granularità nella gestione dei cambiamenti aumentando le performance.

3.5.4 Espressività e flessibilità

Sotto il profilo dell'espressività, React presenta maggiore libertà progettuale in quanto il one-way binding e la combinazione di props, state e hook consentono di

creare qualunque tipo di interazione, nonostante richieda attenzione nel garantire coerenza. Angular rappresenta l'opposto: ha un'architettura più prescrittiva caratterizzata da pattern e convenzioni fortemente standardizzate, che lo rende perfetto per progetti enterprise. Vue adotta una via di mezzo tra queste due filosofie: è abbastanza flessibile da essere adattabile, ma fornisce convenzioni che mitigano la differenziazione stilistica. Svelte, infine, pone maggiore attenzione a semplicità e chiarezza: meno concetti e meno codice da produrre, cosa che significa anche meno controllo su architetture complesse.

3.5.5 Scenari d'uso

Ogni framework risulta più adatto in specifici contesti: Angular eccelle in ambienti enterprise, dove sono essenziali la stabilità e convenzioni forti (come i Reactive Forms). Vue è perfettamente adattabile sia a progetti di scala medio-grande sia a piccole interfacce interattive grazie al suo equilibrio tra semplicità e potenza. Nel caso in cui si abbia a che fare con applicazioni che debbano essere modulari e scalabili e si abbia bisogno di flessibilita e di un ecosistema esteso, React è la soluzione ideale. Infine Svelte è perfetto per applicazioni leggere che richiedono alte prestazioni. Dunque, per riassumere, adotterei le seguenti scelte:

- **Applicazioni enterprise**: Angular (struttura) o React (flessibilità).
- **Team grandi**: React o Angular (standardizzazione).
- **Performance critiche**: Svelte o Vue.
- **Prototipazione rapida**: Vue e Svelte.
- **Progetti piccoli**: Svelte o Vue.

3.5.6 Sintesi conclusiva

Tirando le somme dal capitolo appena concluso, ogni framework interpreta il problema del data binding secondo una propria filosofia, proponendo soluzioni sia conservative che innovative adatte a problemi e ad esigenze progettuali diverse.

Tuttavia, dall'analisi complessiva emerge una chiara tendenza evolutiva dell'ecosistema Javascript, il quale si sta muovendo verso una progressiva riduzione dell'overhead a runtime grazie ad ottimizzazioni *compile-time* e modelli più *fine-grained*. Tutti i framework sono stati chiaramente influenzati dalle innovazioni portate da Svelte con il binding a compile-time, confluendo in novità come il React Compiler e gli Angular Signals.

A questo punto, è evidente che il futuro del data binding sia orientato verso soluzioni sempre più *compiler-driven* e con reattività *fine-grained*, tendenze che saranno oggetto del prossimo capitolo.

Capitolo 4

Verso nuove tendenze e tecnologie

Come accennato nel capitolo 2, dopo aver analizzato in dettaglio lo stato dell'arte dei framework più utilizzati riguardo il binding, in questo capitolo esploreremo i trend più moderni che stanno rimodellando il panorama del data binding. Metteremo a fuoco le innovazioni recenti e parleremo sia della direzione innovativa verso cui si stanno avviando i framework analizzati precedentemente, sia di nuovissimi approcci introdotti da framework meno conosciuti come Qwik e Solid.js.

4.1 Dall'overhead runtime alla reattività predittiva

Negli ultimi anni, l'innovazione nel binding si sta concentrando soprattutto sul decidere in che misura lasciare il lavoro computazionale al runtime e quanto lavoro sia invece possibile delegare alla compilazione. Infatti, abbiamo accennato a come i principali framework moderni stiano evolvendo verso configurazioni che seguono la falsariga di quella di Svelte: il compilatore assume un ruolo centrale nel predeterminare le dipendenze, minimizzando (o eliminando) il diffing e i controlli globali. L'idea centrale è che il binding si sposti da un'attività "continua" ad una mirata e predeterminata e questa transizione è resa possibile dall'adozione e dal miglioramento di tecniche *compiler-driven* e dal ricorso a sistemi reattivi più *fine-grained* che permettano di aggiornare soltanto i nodi del DOM effettivamente interessati e di rendere più prevedibile il comportamento dell'applicazione.

4.2 Innovazioni contemporanee

Di seguito scendiamo nel dettaglio di alcune delle tecnologie più nuove presentate nel mondo del binding.

4.2.1 React Compiler

Rilasciato ad aprile 2025 come Release Candidate (RC), React Compiler è uno strumento di build-time che ottimizza e rende automatiche azioni come memoization, callback e semplificazioni di hook, senza la necessità di apportare modifiche esplicite al codice dell'applicazione [Met25b]. In particolare:

- Il compilatore, attenendosi alle regole di React, analizza il codice staticamente e posiziona la memoization nei punti dove serve maggiormente. [Omo24; Omo25]
- Rende superfluo per lo sviluppatore l'uso di paradigmi come `useMemo`, `useCallback` o `React.memo`, diminuendo di molto il boilerplate [Omo24; Omo25].
- Vengono evitati re-render inutili ottimizzando ogni parte del componente, in modo da ridurre il carico di lavoro lasciato a runtime e migliorare le prestazioni [Omo25].

Il risultato di tutto ciò è che React si avvicina al paradigma "compile-time + reattività predittiva" di Svelte, in cui le decisioni prestazionali non sono più solo nelle mani dello sviluppatore ma sono delegate al compilatore.

4.2.2 Angular Signals

Come abbiamo visto nel capitolo 2, a partire da Angular 16/17, sono stati introdotti i *Signals*, un sistema reattivo che incapsula i valori per renderli osservabili e controllare esattamente le modifiche ad essi apportate. L'incapsulamento permette ad Angular di tracciare le dipendenze del valore osservato e aggiornare solamente le parti interessate dell'UI [Ang25b; Gec23]. La prima versione stabile di Angular con i Signals è la 19, rilasciata nel 2024, ma nel 2025, con la versione 20, abbiamo ufficialmente i Signal come paradigmi integrati alla perfezione e pronti per essere utilizzati nelle nuove applicazioni costruite con Angular.

Le caratteristiche distintive di questo nuovo paradigma sono le seguenti:

- I Signals sono sincroni e seguono un modello pull-based: si registra una dipendenza esplicita ogni volta che si legge un valore di un signal ed ogni volta che avviene una modifica solo i consumatori che lo utilizzano vengono aggiornati; in questo modo si riduce il lavoro di change detection. [Ang25b]
- Non richiedono alcuna gestione manuale: la propagazione interna è gestita automaticamente da Angular. [Gec23]
- Non serve change detection globale perchè gli aggiornamenti avvengono localmente, cosa che porta grandi miglioramenti prestazionali. [Ang25b]

Possiamo fare riferimento all'Esempio 2.1.2 per ricordare il funzionamento di questo nuovo approccio. I signal sono la novità più recente introdotta da Angular, che si concentra sull'ottimizzazione e sull'efficienza del binding più che sullo spostamento verso tecnologie compile-driven [Lea25].

4.2.3 Solid.js e la reattività fine-grained

Solid.js è una libreria JavaScript sviluppata da Ryan Carniato che combina le grandi architetture reattive, già consolidate, di frameworks come React e Vue con modifiche volte ad eliminare i colli di bottiglia nelle performance dei colossi del web developing [Yeh24]. Solid.js si basa interamente su un sistema di Signals simili a quelli di Angular, ma invece di essere una feature, questi ultimi rappresentano le fondamenta dell'architettura del framework. L'idea di fondo è quella di localizzare esattamente le parti dell'interfaccia che dipendono dai valori modificati e aggiornare automaticamente solo quelle. Inoltre, Solid è anche caratterizzato dall'assenza di DOM; dunque, sostanzialmente, sembra un framework come React o Vue, si basa sui signals come Angular, ma funziona senza Virtual DOM come Svelte. Le sue primitive principali sono [Loh24]:

- `createSignal`: per creare una variabile reattiva, un signal;
- `createEffect`: funzione che reagisce ai cambiamenti di uno o più signal e che scatena i side effects;

- `createMemo`: per calcolare valori derivati, simili alle `computed` di Vue. [Awo24; Pet24]

Principali vantaggi:

- Il framework aggiorna direttamente i nodi DOM coinvolti nella modifica del signal osservato, a differenza dei framework più grandi che seguono la gerarchia di componenti per propagare cambiamenti agli stati, cosa che lo rende molto veloce[Bui24; Awo24]
- Dato che solo le porzioni dell'interfaccia direttamente dipendenti dai dati cambiati vengono rielaborate, la granularità è molto alta, garantendo a Solid uno dei posti più alti nella gerarchia dei frameworks a reattività fine-grained. [Pet24]
- Il tracciamento delle dipendenze è implicito: il collegamento tra effetti e signal è registrato automaticamente. [Awo24]
- La sintassi è molto simile a quella di React e la curva di apprendimento è veloce per programmatori che partono conoscendo React e/o Vue.

Principali svantaggi [Dro25]:

- Ecosistema più piccolo;
- Tooling non maturo quanto colossi come React;
- Meno usato per progetti enterprise.

In sostanza, SolidJS, rappresenta un approccio maturo al paradigma "reattività minimale + compilazione ottimizzata", un'alternativa leggera e veloce a framework più popolari e una dimostrazione dell'efficacia di architetture a reattività a granularità molto fine.

4.2.4 Resumability con Qwik

Qwik è un framework molto recente, rilasciato in versione stabile nel 2023, che si propone come un'evoluzione dei modelli di hydration e di SSR tradizionali grazie

all'introduzione del concetto di `resumability` . La resumability consiste nel mettere in pausa il rendering dell'app sul server e farlo ripartire (*resume*) direttamente sul client, riprendendo lo stato già renderizzato senza dover re-idratare l'intera applicazione [Qwi25b]. Scendendo più nel dettaglio, Qwik non richiede nessun tipo di hydration dei componenti durante il rendering client-side, nè ha bisogno di ricollegarne l'interattività, perchè, nella maggior parte dei casi d'uso, non esegue proprio i componenti sul client, ma questi rimangono sul server. Infatti, la resumability serve proprio come strumento per permettere al framework di riprendere il suo stato server senza rieseguire i componenti sul client, in modo che quest'ultimo possa evitare il costo di ricalcolare l'albero dei componenti [Hev22].

Aspetti chiave:

- Sul server viene costruito il grafo di reattività, che poi viene serializzato nell'HTML della pagina. in questo modo il client può riprendere esattamente lo stato precedente senza necessità di re-eseguire il binding;
- All'interazione dell'utente con l'UI, Qwik scarica solamente il modulo corrispondente all'handler richiesto evitando l'esecuzione di JavaScript non necessario in anticipo. [Qwi25b]
- La reattività di Qwik è fine-grained e il framework utilizza signals: quando un signal cambia, invalida solo i componenti che lo consumano senza farne il re-render nella loro interezza. [Qwi25b]
- Il binding in Qwik avviene tramite `useSignal()` per valori primitivi e `useStore()` per oggetti complessi. La lettura di un signal nel template registra una dipendenza e ogni modifica aggiorna solo i nodi dipendenti in modo mirato [Qwi25a].

Principali vantaggi [Dro25]:

- No hydration;
- Caricamento istantaneo;
- Performance molto elevate grazie alla resumability;

- Reattività fine-grained;
- Ideale per scenari applicativi grandi.

Principali svantaggi [Dro25]:

- Ecosistema e community giovani e in evoluzione.
- Tooling ancora in maturazione;
- Meno risorse didattiche disponibili;
- Curva di apprendimento ripida.

Per concludere, Qwik si presenta come un'alternativa rivoluzionaria che riscrive il modo in cui viene concepito lo sviluppo web e che mette in gioco un approccio unico per elevare al massimo le performance applicative. È un framework nato per produrre grande efficienza e vale la pena seguirne gli sviluppi futuri.

4.2.5 Tecnologie a confronto

Di seguito, proponiamo uno specchietto comparativo delle tecnologie menzionate in questo capitolo:

Criterio	React	Angular	Solid.js	Qwik
Velocità	Migliore grazie a React Compiler, dipende ancora dal Virtual DOM	Migliore grazie ai Signals	Molto alta: grazie ad aggiornamenti diretti ed assenza di Virtual DOM	Massima grazie alla resumability
Efficienza	Grazie alle ottimizzazioni automatiche compile-time è ottima	Eccellente in progetti complessi, grazie a change detection mirata	Eccellente: no overhead runtime	Eccezionale grazie alla minima esecuzione di JS sul client
Sintassi / Ergonomia	Esplicita e consolidata, ancora un po' verbosa	Strutturata e rigorosa	Simile a React, più naturale e concisa	Modulare, un po' meno immediata rispetto agli altri
Scalabilità	Ecosistema consolidato	Eccellente per applicazioni enterprise	Ecosistema ridotto	Ecosistema giovane
Maturità	Altissima, supporto e community enormi	Molto alta, supporto ampio e consolidato	Media, la community è ancora in crescita	Ancora da consolidare

Tabella 4.1: Confronto sintetico tra i principali framework di nuova generazione.

4.3 Prospettive e traiettorie evolutive

Anche se i paradigmi analizzati in questo capitolo sono ancora in fase di adozione, possiamo delineare degli orientamenti comuni destinati a trainare il futuro del binding:

- **Compilatori incrementalmente più intelligenti**: si va sempre di più verso l'automatizzazione della logica di binding spostando l'onere verso il compilatore piuttosto che sul runtime così da ridurre la necessità di scrivere codice "proattivo" per l'ottimizzazione;
- **Reattività contestuale e predittiva**: ci si sposta verso sistemi sempre più in grado di prevedere quali nodi dovranno essere aggiornati a seguito di una certa modifica di stato e quando, analizzando il grafo delle dipendenze.
- **Framework compositi e ibridi**: i framework si muoveranno sempre di più verso approcci ibridi da applicare in parti diverse dell'applicazione: reattività fine-grained per le aree dinamiche e componenti resume/hydrate per sezioni meno interattive.
- **Minimizzazione del JavaScript eseguito lato client**: resumability e ottimizzazione a compile-time sono paradigmi che riducono drasticamente il codice attivo nel browser, aumentando di molto le performance, specialmente su dispositivi deboli e meno efficienti.

In definitiva, l'idea alla base di tutte queste tendenze evolutive è che il binding non sarà più solo un problema di "aggiornare model e view quando uno stato cambia", ma anche di integrare la reattività direttamente nel processo di compilazione, diminuendo il numero e la necessità dei re-rendering e puntando a prevedere dove e quando debba essere eseguito. Tutto ciò si può dire che elevi il data-binding quasi a principio architetturale.

Conclusioni

Questa tesi ha messo in evidenza come il *data binding* rappresenti, ancora oggi, un problema aperto tra i più complessi e determinanti nella progettazione di applicazioni web moderne. Abbiamo, infatti, visto come le modalità con cui i vari framework implementano la sincronizzazione di modello e vista riflettano filosofie estremamente diverse in termini di gestione dello stato, efficienza e reattività.

Siamo partiti dal definire il concetto di data-binding, le sue caratteristiche e le problematiche ad esso legate, che ci hanno portato a comprendere quando sia importante analizzare da vicino i paradigmi di implementazione più usati per poter decidere quale tecnologia è più adatta alle nostre esigenze.

Dal confronto tra *Angular*, *React*, *Vue* e *Svelte* è emerso, dunque, che non esiste un vero e proprio vincitore, quanto piuttosto una serie di approcci che si propongono di trovare compromessi in funzione delle priorità progettuali. Abbiamo scoperto che Angular, grazie all'architettura fortemente strutturata e all'introduzione dei *Signals*, coniuga brillantemente stabilità e chiarezza sintattica, ma a costo di una maggiore complessità nell'apprendimento e nel padroneggiamento dei concetti. React, invece, è incentrato sul flusso unidirezionale e sul *Virtual DOM* e questo permette di aumentare la prevedibilità e la scalabilità, richiedendo però un maggiore difficoltà nella gestione manuale dello stato. Vue offre una sintesi intermedia che unisce semplicità nella dichiarazione delle entità e reattività fine-grained, rappresentando una valida alternativa più intuitiva per contesti di dimensioni non troppo elevate. Infine abbiamo dedotto che Svelte si pone come l'avanguardia del compile-time binding, poichè elimina il peso del *diffing* ed è in grado di anticipare la logica reattiva in fase di compilazione.

Grazie al confronto condotto su sintassi, espressività ed efficienza abbiamo messo

in evidenza come reattività fine-grained e tecniche di ottimizzazione a compile-time rappresentino ormai la frontiera più promettente per ridurre l'overhead a runtime e favorire sia la chiarezza che la performance. Le analisi dell'ultimo capitolo, infatti, confermano che l'interesse della comunità si sta spostando verso modelli che cercano di migliorare il comportamento dell'applicazione prima ancora che essa venga eseguita, in modo da bilanciare maggiormente l'interazione tra sviluppatore, framework e macchina.

In termini di espressività, si osserva che i framework più moderni sono volti ad offrire un linguaggio di binding sempre più naturale e intuitivo, che si pone di descrivere relazioni complesse in modo più breve senza sacrificare la leggibilità del codice.

Infine, concentrandoci sull'efficienza, il passaggio da modelli a controllo imperativo a quelli a controllo reattivo ha fatto in modo che i costi computazionali fossero ridotti drasticamente, migliorando la scalabilità in applicazioni di grandi dimensioni.

Un aspetto che ho trovato particolarmente interessante durante la scrittura di questo elaborato è stato il concetto di "resumability" in Qwik, che sembra un concetto banale, ma in realtà rappresenta un modo di pensare al binding nuovo e fuori dagli schemi, che non si ferma a migliorare le basi già consolidate da framework più vecchi.

In conclusione, il lavoro intrapreso in questa dissertazione evidenzia come il *data binding* non sia un problema definitivamente risolto, ma un ambito in continua evoluzione, del quale si continuerà sicuramente a parlare nel prossimo futuro. I framework analizzati propongono differenti risposte a un'unica sfida: quella di coniugare semplicità, efficienza e chiarezza in un ecosistema in rapido mutamento.

Proprio la comprensione delle differenze tra questi approcci ci porta, quindi, non solo a scegliere lo strumento più adatto, ma anche a cogliere l'evoluzione del pensiero che guida lo sviluppo web contemporaneo. Ed è in questo concetto che risiede l'idea iniziale della mia ricerca: mostrare che dietro la sintassi e i meccanismi tecnici del binding si nasconde, in realtà, un'intera visione dell'interazione tra l'uomo, il codice e la tecnologia.

Bibliografia

[Ang] AngularJS Team. Expressions (one-time binding). https://docs.angularjs.org/guide/expression. Ultima consultazione: 9 ottobre 2025.

[Ang25a] Angular Team. Angular forms guide. https://angular.dev/guide/forms, 2025.

[Ang25b] Angular Team. Angular signals: Reactivity model. https://angular.dev/guide/signals, 2025.

[Ang25c] Angular Team. Changedetectionstrategy – angular api. https://angular.dev/api/core/ChangeDetectionStrategy, 2025.

[Ang25d] Angular Team. Template syntax. https://angular.dev/guide/templates/binding, 2025.

[Awo24] Chris Awoke. Reactivity in solid.js: How fine-grained reactivity really works. https://blog.openreplay.com/reactivity-in-solid/, 2024. Articolo pubblicato su *OpenReplay blog*.

[Bui23] Builder.io Team. Qwik city: Server functions and progressive hydration. https://www.builder.io/blog/qwik-city-server-functions, 2023.

[Bui24] Builder.io. Reactivity across frameworks. https://www.builder.io/blog/reactivity-across-frameworks, 2024.

[Dha24] Hiren Dhaduk. Angular vs vue: which framework to choose in 2024? https://www.simform.com/blog/angular-vs-vue/, 2024.

[Dro25] Eleftheria Drosopoulou. Qwik vs react vs solidjs: The future of web performance. https://www.javacodegeeks.com/2025/06/qwik-vs-react-vs-solidjs-the-future-of-web-performance.html, 2025.

[Gec23] Minko Gechev. Angular signals are here. https://blog.angular.dev/angular-v16-is-here-4d7a28ec680d, 2023.

[Har18] Rich Harris. Virtual dom is pure overhead. https://svelte.dev/blog/virtual-dom-is-pure-overhead, 2018.

[Hev22] Miško Hevery. Builder resumability vs hydration. https://www.builder.io/blog/resumability-vs-hydration, 2022. Articolo pubblicato su *Builder.io*.

[Kle23] Vladimir Klepov. Svelte state: an inside and out guide to svelte reactivity. https://thoughtspile.github.io/2023/04/22/svelte-state/, 2023.

[Koc23] Oleksandr Kocherhin. Data binding in angular – one-way and two-way binding explained. https://www.youtube.com/watch?v=yV5bw-MZgIk, 2023. Video pubblicato sul canale YouTube *Monsterlessons Academy*.

[Kra24] Nimrod Kramer. Svelte compiler: How it works. https://daily.dev/blog/svelte-compiler-how-it-works, 2024. Articolo pubblicato sulla piattaforma *daily.dev*.

[Lea25] Learn With Awais. Why angular v20's new two-way binding is a game changer for uis. https://learnwithawais.medium.com/why-angular-v20s-new-two-way-binding-is-a-game-changer-for-uis-6a6e2f239842, 2025. `Learn_With_Awais` è lo username dell'autore dell'articolo pubblicato su *Medium*.

[Loh24] Alex Lohr. Primitive di solid.js. https://dev.to/lexlohr/the-zen-of-state-in-solidjs-22lj, 2024. Articolo pubblicato su *Dev community: dev.to*.

[Met25a] Meta Platforms Inc. Forms – react dom components. https://react.dev/reference/react-dom/components/form, 2025.

[Met25b] Meta Platforms Inc. Introducing react compiler. https://react.dev/learn/react-compiler/introduction, 2025.

[Met25c] Meta Platforms Inc. React documentation: usememo. https://react.dev/reference/react/useMemo, 2025.

[Met25d] Meta Platforms Inc. React hooks – react documentation. https://react.dev/reference/react/hooks, 2025.

[Met25e] Meta Platforms Inc. React official documentation. https://react.dev/, 2025.

[Mic25] Microsoft Corporation. Data binding overview – windows presentation foundation (wpf). https://learn.microsoft.com/en-us/dotnet/desktop/wpf/data/, 2025.

[Mor25] Eduardo San Martin Morote. Pinia introduction. https://pinia.vuejs.org/introduction.html, 2025.

[Nux25] Nuxt - MIT License. Rendering modes — nuxt concepts. https://nuxt.com/docs/guide/concepts/rendering, 2025.

[Omo24] David Omotayo. Exploring the react compiler: a detailed introduction. https://blog.logrocket.com/exploring-react-compiler-detailed-introduction/, 2024.

[Omo25] David Omotayo. Exploring the react compiler (rc). https://blog.logrocket.com/react-compiler-rc/, 2025.

[Pet24] Rosen Petkov. Reactivity as the catalyst driving evolution in frameworks like react, vue, svelte and angular. https://medium.com/@dadcod/reactivity-as-the-catalyst-driving-evolution-in-frameworks-like-react-vue-svelte-and-angular 2024.

[Qwi25a] Qwik Team. Getting started with qwik. https://qwik.dev/docs/getting-started/, 2025.

[Qwi25b] Qwik Team. Qwik resumability vs hydration. https://qwik.dev/docs/concepts/resumable/, 2025.

[Ras25] Rasla. React's new compiler is a game changer — bye memo, hello auto-magic. https://medium.com/@raslarasla/reacts-new-compiler-is-a-game-changer-bye-memo-hello-auto-magic-36ad4021beae, 2025. Articolo pubblicato con username "Lazy Code" sul blog *Medium*.

[Ric21] Geoff Rich. What does $ mean in svelte? https://geoffrich.net/posts/svelte-$-meanings/, 2021.

[Sch24] Maximilian Schwarzmüller. React vs angular vs vue vs svelte – frontend framework comparison. https://www.youtube.com/watch?v=K2lvjqvpajc, 2024.

[Sof24] Software Mind. Vue vs react: A comprehensive comparison for modern development. https://softwaremind.com/blog/vue-vs-react-a-comprehensive-comparison-for-modern-development/, 2024.

[Sve25a] Svelte contributors. Svelte official docs. https://svelte.dev/docs/svelte/overview, 2025.

[Sve25b] Svelte contributors. Svelte store. https://svelte.dev/docs/svelte/svelte-store, 2025.

[Sve25c] Svelte Contributors. Sveltekit documentation – load function: Streaming. https://svelte.dev/docs/kit/load#streaming, 2025.

[Sve25d] Svelte Contributors. Sveltekit documentation – project types. https://svelte.dev/docs/kit/project-types, 2025.

[Sve25e] Svelte Maintainers. bind: directive. https://svelte.dev/docs/svelte/bind, 2025.

[Tec23] Technocrat. Comparing javascript frameworks: Svelte, vue, react, and angular. https://medium.com/coderhack-com/

comparing-javascript-frameworks-svelte-vue-react-and-angular-70474666e69d, 2023. Articolo pubblicato su *CoderHack.com*.

[Ver25] Vercel Inc. Next.js documentation. https://nextjs.org/docs, 2025.

[Vou17] Jaakko Voutilainen. Comparison of data binding techniques in front-end development. https://www.theseus.fi/bitstream/handle/10024/138668/Voutilainen_Jaakko.pdf?sequence=1&isAllowed=y, 2017.

[Vue25a] Vue.js Core Team. Built-in directives. https://vuejs.org/api/built-in-directives.html, 2025.

[Vue25b] Vue.js Core Team. Computed properties – reactivity fundamentals. https://vuejs.org/guide/essentials/computed.html, 2025.

[Vue25c] Vue.js Core Team. Form input bindings. https://vuejs.org/guide/essentials/forms, 2025.

[Vue25d] Vue.js Core Team. Form input modifiers. https://vuejs.org/guide/essentials/forms.html#modifiers, 2025.

[Vue25e] Vue.js Core Team. Reactivity fundamentals. https://vuejs.org/guide/essentials/reactivity-fundamentals.html, 2025.

[Vue25f] Vue.js Core Team. Rendering mechanism. https://vuejs.org/guide/extras/rendering-mechanism, 2025.

[Vue25g] Vue.js Core Team. Server-side rendering — vue. https://vuejs.org/guide/scaling-up/ssr.html, 2025.

[Vue25h] Vue.js Core Team. Template syntax – vue reactivity fundamentals. https://vuejs.org/guide/essentials/template-syntax.html, 2025.

[Vue25i] Vue.js Core Team. Template syntax – vue reactivity fundamentals. https://vuejs.org/api/composition-api-dependency-injection.html, 2025.

[Vue25j] Vue.js Core Team. Vue.js security guide. https://vuejs.org/guide/best-practices/security.html, 2025.

[Wik25a] Wikipedia contributors. Virtual dom. https://en.wikipedia.org/wiki/Virtual_DOM, 2025.

[Wik25b] Wikipedia contributors. Vue.js — wikipedia. https://en.wikipedia.org/wiki/Vue.js, 2025.

[Yeh24] Raymond Yeh. What is solid.js – understanding the modern reactive library. https://www.wisp.blog/blog/what-is-solidjs-understanding-the-modern-reactive-library, 2024. Articolo pubblicato su *Wisp.blog*.

[You25] Evan You. Vue.js documentation. https://vuejs.org/, 2025.