# Module 2: Frontend Single Page Applications

## React Core, Forms & Routing

**Adrián Catalán**

adriancatalan@galileo.edu

# Agenda

1. **Project: Real Estate React**
2. **React Core (Vite) & Hooks**
3. **Forms (Zod + React Hook Form)**
4. **Routing & LocalStorage**
5. **Deep Dive**
6. **Challenge Lab**

# Real Estate App

We are building a Real Estate listing application with React.

**Core Requirements:**

1. **Listing**: Display property cards with images and details.
2. **Filtering**: Filter by type, price range, and search.
3. **Forms**: Add new properties with validation.
4. **Navigation**: Multi-page app with React Router.


center

# 2. React Basics

# What is React?

React is a JavaScript library for building user interfaces.

**Key Concepts:**

1. **Declarative**: Describe what UI should look like, not how to build it.
2. **Component-Based**: Build encapsulated components that manage their own state.
3. **Virtual DOM**: React updates only what changed, not the entire page.

```
// Declarative - What, not how
function Greeting({ name }) {
    return <h1>Hello, {name}!</h1>;
}
```

# JSX: JavaScript + HTML

JSX is a syntax extension that looks like HTML in JavaScript.

```
// JSX (What you write)
const element = <h1 className="title">Hello, World!</h1>;

// JavaScript (What it compiles to)
const element = React.createElement(
    'h1',
    { className: 'title' },
    'Hello, World!'
);
```

### Key Differences from HTML:

- `className` instead of `class`
- `htmlFor` instead of `for`
- CamelCase for attributes (`onClick`, `onChange`)

# Components: The Building Blocks

Components are reusable pieces of UI.

```
// Function Component (Modern)
function PropertyCard({ property }: { property: Property }) {
    return (
        <div className="card">
            <img src={property.imageUrl} alt={property.title} />
            <h2>{property.title}</h2>
            <p>${property.price.toLocaleString()}</p>
        </div>
    );
}

// Usage
<PropertyCard property={myProperty} />
```

# Props: Passing Data Down

Props are read-only inputs to components.

```
// Parent component
function App() {
    return (
        <PropertyCard
            title="Beach House"
            price={500000}
            bedrooms={3}
        />
    );
}

// Child component
interface PropertyCardProps {
    title: string;
    price: number;
    bedrooms: number;
}

function PropertyCard({ title, price, bedrooms }: PropertyCardProps) {
    return <div>{title} - ${price} - {bedrooms} beds</div>;
```

# Children: Composition

The `children` prop allows component composition.

```
// Reusable Card wrapper
function Card({ children }: { children: React.ReactNode }) {
    return (
        <div className="bg-white rounded-lg shadow-md p-4">
            {children}
        </div>
    );
}

// Usage
<Card>
    <h2>Property Title</h2>
    <p>Description here...</p>
    <Button>View Details</Button>
</Card>
```

# Conditional Rendering

Render different UI based on conditions.

```
function PropertyCard({ property }: Props) {
    return (
        <div className="card">
            {/* Conditional with && */}
            {property.featured && <Badge>Featured</Badge>}

            {/* Ternary operator */}
            {property.available
                ? <span className="text-green-500">Available</span>
                : <span className="text-red-500">Sold</span>
            }

            {/* Early return pattern */}
            {!property.imageUrl && <Placeholder />}
        </div>
    );
}
```

# Lists & Keys

Render arrays of data with `.map()`.

```
function PropertyList({ properties }: { properties: Property[] }) {
    return (
        <div className="grid grid-cols-3 gap-4">
            {properties.map((property) => (
                <PropertyCard
                    key={property.id}  // Required unique identifier
                    property={property}
                />
            ))}
        </div>
    );
}
```

**Why Keys?** React uses keys to identify which items changed, were added, or removed.

# 3. Hooks & State

# useState: Component State

State is data that can change over time.

```
import { useState } from 'react';

function Counter() {
    // Declare state variable with initial value
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>
                Increment
            </button>
            <button onClick={() => setCount(prev => prev - 1)}>
                Decrement (functional update)
            </button>
        </div>
    );
}
```

# State Updates are Asynchronous

React batches state updates for performance.

```
// BAD: May not work as expected
function handleClick() {
    setCount(count + 1);
    setCount(count + 1);
    // Result: count + 1, not count + 2
}

// GOOD: Use functional updates
function handleClick() {
    setCount(prev => prev + 1);
    setCount(prev => prev + 1);
    // Result: count + 2
}
```

**Rule:** When new state depends on previous state, use the functional form.

# Complex State with Objects

```
interface Filters {
    search: string;
    type: string;
    maxPrice: number;
}

function PropertyFilters() {
    const [filters, setFilters] = useState<Filters>({
        search: '',
        type: 'all',
        maxPrice: 1000000
    });

    const updateFilter = (key: keyof Filters, value: string | number) => {
        setFilters(prev => ({
            ...prev,          // Spread previous state
            [key]: value      // Update specific field
        }));
    };

    return <input onChange={(e) => updateFilter('search', e.target.value)} />;
}
```

# useEffect: Side Effects

Effects let you run code after render.

```
import { useEffect, useState } from 'react';

function PropertyList() {
    const [properties, setProperties] = useState<Property[]>([]);

    useEffect(() => {
        // Runs after every render (no dependency array)
        // Runs once on mount (empty array [])
        // Runs when dependencies change ([dep1, dep2])

        async function fetchProperties() {
            const data = await api.getProperties();
            setProperties(data);
        }

        fetchProperties();
    }, []); // Empty array = run once on mount

    return <div>{/* render properties */}</div>;
```

# Effect Cleanup

Some effects need cleanup (subscriptions, timers).

```
useEffect(() => {
    // Setup
    const timer = setInterval(() => {
        console.log('Tick');
    }, 1000);

    // Cleanup function (runs before next effect or unmount)
    return () => {
        clearInterval(timer);
    };
}, []);

// With event listeners
useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener('resize', handleResize);

    return () => window.removeEventListener('resize', handleResize);
```

# useMemo: Expensive Calculations

Memoize computed values to avoid recalculation.

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

function PropertyList({ properties, filters }: Props) {
    // Only recalculates when properties or filters change
    const filteredProperties = useMemo(() => {
        return properties
            .filter(p => p.type === filters.type || filters.type === 'all')
            .filter(p => p.price <= filters.maxPrice)
            .filter(p => p.title.toLowerCase().includes(filters.search));
    }, [properties, filters]);

    return (
        <div>
            {filteredProperties.map(p => <PropertyCard key={p.id} property={p} />)}
        </div>
    );
}
```

# useCallback: Stable Function References

Memoize functions to prevent unnecessary re-renders.

```
import { useCallback } from 'react';

function PropertyList() {
    const [properties, setProperties] = useState<Property[]>([]);

    // Without useCallback, this creates a new function every render
    // causing child components to re-render unnecessarily
    const handleDelete = useCallback((id: string) => {
        setProperties(prev => prev.filter(p => p.id !== id));
    }, []); // No dependencies = stable reference

    return (
        <div>
            {properties.map(p => (
                <PropertyCard key={p.id} property={p} onDelete={handleDelete} />
            ))}
        </div>
    );
}
```

# 4. Component Patterns

# Shadcn UI: Component Library

Pre-built, customizable components based on Radix UI.

```
# Installation
npx shadcn@latest init
npx shadcn@latest add button card input
```

```
// Usage
import { Button } from '@/components/ui/button';
import { Card, CardHeader, CardTitle, CardContent } from '@/components/ui/card';

function PropertyCard() {
    return (
        <Card>
            <CardHeader>
                <CardTitle>Beach House</CardTitle>
            </CardHeader>
            <CardContent>
                <Button variant="outline">View Details</Button>
            </CardContent>
        </Card>
```

# Form Handling with React Hook Form

Uncontrolled forms with validation using React Hook Form + Zod.

```
import { useForm } from 'react-hook-form';
import { createPropertySchema, CreatePropertyInput } from '@/types/property';

function PropertyForm({ onSubmit }: Props) {
    const { register, handleSubmit, formState: { errors } } = useForm<CreatePropertyInput>({
        resolver: async (values) => {
            const result = createPropertySchema.safeParse(values);
            if (result.success) return { values: result.data, errors: {} };

            const errors = result.error.issues.reduce((acc, issue) => ({
                ...acc,
                [issue.path[0]]: { type: issue.code, message: issue.message }
            }), {});
            return { values: {}, errors };
        },
        defaultValues: { title: '', price: 0, propertyType: 'apartamento' }
    });

    return (
        <form onSubmit={handleSubmit(onSubmit)}>
            <input {...register('title')} />
            {errors.title && <p>{errors.title.message}</p>}
            <button type="submit">Save</button>
        </form>
```

# Form Validation

Validate inputs before submission.

```
function PropertyForm() {
    const [errors, setErrors] = useState<Record<string, string>>({});

    const validate = (): boolean => {
        const newErrors: Record<string, string> = {};

        if (!formData.title.trim()) {
            newErrors.title = 'Title is required';
        }
        if (Number(formData.price) <= 0) {
            newErrors.price = 'Price must be positive';
        }

        setErrors(newErrors);
        return Object.keys(newErrors).length === 0;
    };

    const handleSubmit = (e: React.FormEvent) => {
        e.preventDefault();
        if (validate()) {
            // Submit form
        }
```

# React Router: Navigation

Client-side routing for SPAs.

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

function App() {
    return (
        <BrowserRouter>
            <nav>
                <Link to="/">Home</Link>
                <Link to="/new">Add Property</Link>
            </nav>

            <Routes>
                <Route path="/" element={<HomePage />} />
                <Route path="/new" element={<NewPropertyPage />} />
                <Route path="/property/:id" element={<PropertyDetailPage />} />
                <Route path="*" element={<NotFound />} />
            </Routes>
        </BrowserRouter>
    );
}
```

# Dynamic Routes & Params

Access URL parameters in components.

```
import { useParams, useNavigate } from 'react-router-dom';
import { getPropertyById } from '@/lib/storage';

function PropertyDetailPage() {
    const { id } = useParams<{ id: string }>();
    const navigate = useNavigate();
    const [property, setProperty] = useState<Property | null>(null);

    useEffect(() => {
        if (id) {
            const found = getPropertyById(id);
            setProperty(found);
        }
    }, [id]);

    if (!property) return <p>Property not found</p>;

    return (
        <div>
            <h1>{property.title}</h1>
            <button onClick={() => navigate('/')}>Go Home</button>
        </div>
```

# Local Storage Persistence

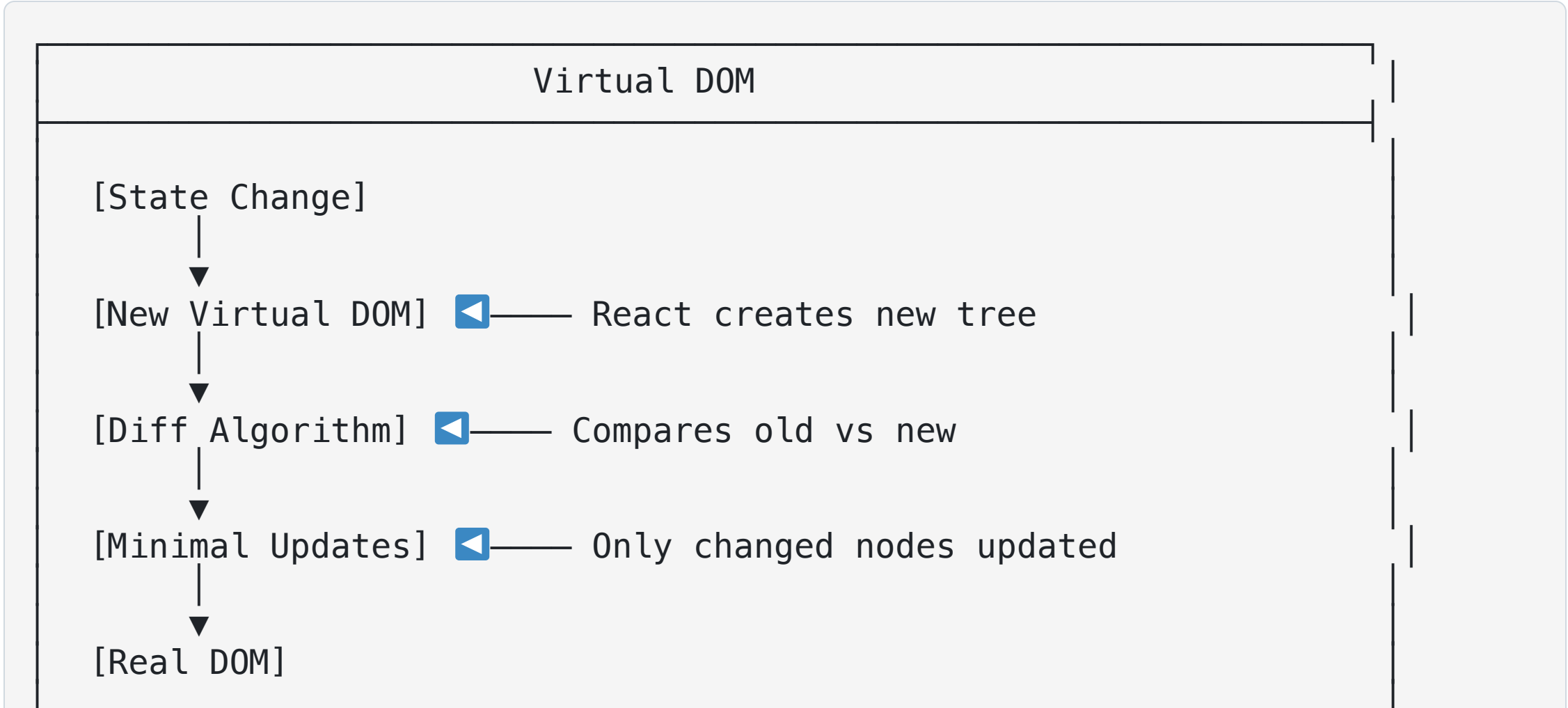
Save state to localStorage for persistence.

```
// lib/storage.ts
export function saveToStorage<T>(key: string, data: T): void {
    localStorage.setItem(key, JSON.stringify(data));
}

export function loadFromStorage<T>(key: string, defaultValue: T): T {
    const stored = localStorage.getItem(key);
    return stored ? JSON.parse(stored) : defaultValue;
}

// Usage in component
const [properties, setProperties] = useState<Property[]>(() =>
    loadFromStorage('properties', sampleProperties)
);

useEffect(() => {
    saveToStorage('properties', properties);
```

# 5. Deep Dive

# 1. Virtual DOM

React uses a virtual representation of the DOM for efficiency.



# 2. Reconciliation & Keys

How React decides what to update.

### Without Keys:

```
// React can't tell which item changed
<ul>
    <li>Item A</li>
    <li>Item B</li>  {/* If removed, React might update wrong item */}
    <li>Item C</li>
</ul>
```

### With Keys:

```
// React tracks each item by key
<ul>
    {items.map(item => (
        <li key={item.id}>{item.name}</li>  // Stable identity
    ))}
```

# 3. React Strict Mode

Development tool for finding potential problems.

```
// main.tsx
import { StrictMode } from 'react';

createRoot(document.getElementById('root')!).render(
    <StrictMode>
        <App />
    </StrictMode>
);
```

**What it does:**

- Renders components twice (in dev) to catch side effects
- Warns about deprecated APIs
- Detects unexpected side effects in render phase

# 4. Component Re-rendering

Understanding when components re-render.

**Triggers:**

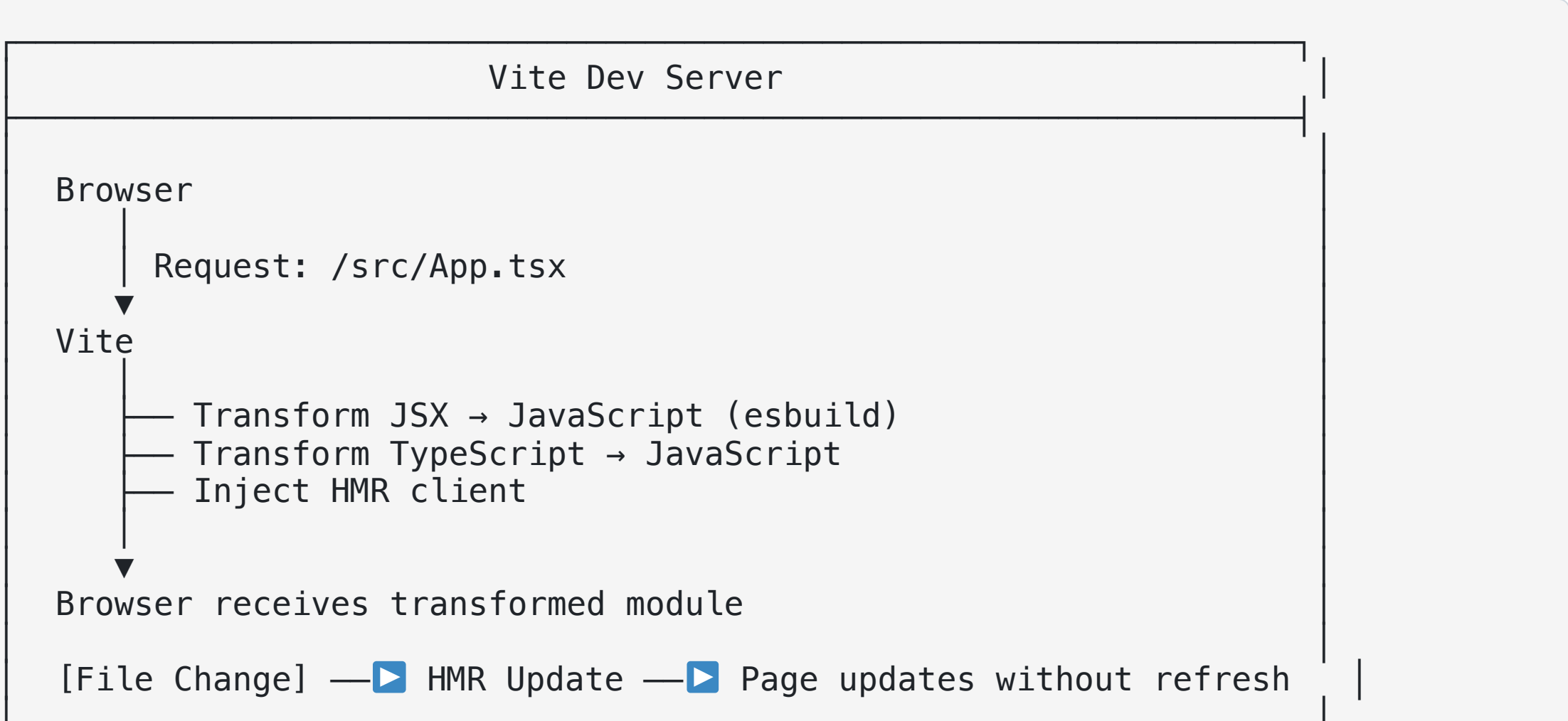
1. State changes (`setState`)
2. Props change (parent re-renders)
3. Context value changes

**Optimization:**

```
// React.memo prevents re-render if props unchanged
const PropertyCard = React.memo(function PropertyCard({ property }: Props) {
    return <div>{property.title}</div>;
});

// Only re-renders if property object reference changes
```

# 5. Vite + React

How Vite serves React applications.



# 6. Challenge Lab

## Practice & Application

# Part 1: Property Comparison

**Context:**

Users want to compare multiple properties side-by-side before making a decision.

**Your Task:**

Implement a comparison feature that:

- Allows selecting up to 3 properties for comparison
- Shows a comparison table with key metrics
- Highlights the best value in each category
- Provides a way to remove properties from comparison

**Files to Create/Modify:**

- `src/components/CompareButton.tsx`
- `src/pages/ComparePage.tsx`

# Part 1: Definition of Done

| Criteria | Description |
|---|---|
| Selection UI | Checkbox or button to select properties |
| Max 3 limit | Cannot select more than 3 properties |
| Comparison table | Side-by-side view with property details |
| Metrics shown | Price, bedrooms, bathrooms, area, price/sqm |
| Best highlighted | Lowest price, highest area highlighted |
| Remove from compare | Can deselect from comparison view |
| Empty state | Message when no properties selected |

# Part 2: Image Gallery

**Context:**

Properties should support multiple images with a gallery view on the detail page.

**Your Task:**

Implement an image gallery that:

- Shows thumbnail grid of all property images
- Opens full-screen modal on click
- Supports keyboard navigation (arrow keys, escape)
- Shows image counter (e.g., "3 of 10")

**Files to Create/Modify:**

- `src/components/ImageGallery.tsx`
- `src/components/ImageModal.tsx`

## Part 2: Definition of Done

| Criteria | Description |
|---|---|
| Thumbnails grid | Property images shown as clickable thumbnails |
| Modal opens | Clicking thumbnail opens full-screen view |
| Navigation arrows | Left/right buttons to navigate images |
| Keyboard support | Arrow keys and Escape work |
| Image counter | Shows current position (e.g., "3 of 10") |
| Close button | X button to close modal |
| Backdrop click | Clicking outside image closes modal |

# Resources & Wrap-up

# Resources

## React

- React Documentation
- React Tutorial (Tic-Tac-Toe)
- Thinking in React
- React Hooks Reference

## TypeScript + React

- React TypeScript Cheatsheet
- Total TypeScript: React

## Shadcn UI

- Shadcn UI Documentation

# Recommended Articles

## React Fundamentals

- A Complete Guide to useEffect - Dan Abramov
- Before You memo() - Dan Abramov
- React re-renders guide - Developer Way

## Performance

- Optimizing Performance - React Docs
- useMemo and useCallback - Kent C. Dodds
- When to useMemo and useCallback - Kent C. Dodds