React & Next.js Learning Notes

From Building My Interactive CV - October 2025

Table of Contents

Stage 1 - Core Concepts

Stage 2 - Advanced Interactions

1. Components & JSX

What is a Component?

A component is a **reusable piece of UI** - it's just a JavaScript function that returns JSX.

```
// Simple component
function Greeting() {
  return <h1>Hello World!</h1>
}

// Use it like HTML
<Greeting />
```

Components in Vanilla React vs Next.js

- **Components ARE vanilla React** - they're React's core feature
- **Next.js adds** file-based routing, server components, optimizations
- **The component syntax is identical** in both

What is JSX?

JSX = JavaScript XML (NOT JSON!)

```
// This looks like HTML but it's JavaScript
const element = <h1 className="title">Hello</h1>

// React converts it to:
const element = React.createElement('h1', { className: 'title' }, 'Hello')
```

Key differences from HTML:

- Use `className` instead of `class`
- Use `htmlFor` instead of `for`
- Self-closing tags need `/` (e.g., `<img />`)
- JavaScript expressions go in curly braces: `{variable}`

Who Decides Where to Render?

The Rendering Tree:

```
app/layout.tsx (Root - wraps everything)
    ↓
app/page.tsx (Main page - assembles sections)
    ↓
components/Hero.tsx (Individual section)
```

Example:

```
// app/page.tsx - YOU decide the order
export default function Home() {
  return (
    <div>
      <Navigation />  {/* Rendered first */}
      <Hero />        {/* Then this */}
      <Experience />  {/* Then this */}
    </div>
  )
}
```

React handles the actual DOM updates - you just declare what should be shown.

---

2. Props - Passing Data

What Are Props?

Props (properties) are how you pass data to components - **like function parameters**.

Props Can Be ANY Data Type:

```
// String
<Hero name="John" />

// Number
<Hero age={25} />

// Boolean
<Hero isActive={true} />

// Object
<Hero person={{ name: "John", age: 25 }} />

// Array
<Hero skills={["React", "TypeScript"]} />

// Function
<Hero onClick={() => alert("Hi")} />
```

Real Example from Your CV:

```
// Passing an object as a prop
<ExperienceCard experience={exp} />

// Receiving the prop (destructuring syntax)
function ExperienceCard({ experience }) {
  // Access properties:
  return (
    <div>
      <h3>{experience.company}</h3>
      <p>{experience.position}</p>
    </div>
  )
}
```

Props are Just Function Parameters:

```
// Regular function
function greet(name) {
  return "Hello " + name
}
greet("John")

// React component - same concept!
function Greeting({ name }) {
```

```
  return <h1>Hello {name}</h1>
}
<Greeting name="John" />
```

Iterating Over Array Props:

```
function SkillsList({ skills }) {
  return (
    <div>
      {skills.map((skill) => (
        <span key={skill}>{skill}</span>
      ))}
    </div>
  )
}

// Usage:
<SkillsList skills={["React", "TypeScript", "Node.js"]} />

// Renders:
<div>
  <span>React</span>
  <span>TypeScript</span>
  <span>Node.js</span>
</div>
```

---

3. State - Managing Changes

What is State?

State is **data that can change over time** and causes the component to re-render when it changes.

Do I Need State? Decision Tree:

```
Does clicking/interacting change visible data on the page?
├─ NO → Don't use state (just use regular HTML/links)
│       Examples: Links, "Back to top" button, external links
│
└─ YES → Use state!
         Examples: Filters, counters, form inputs, toggles
```

Does NOT Need State:

```
// Just navigation - no data changes
<a href="#top">Back to Top</a>
```

```
<Link href="#about">About</Link>
<a href="https://github.com">GitHub</a>
```

DOES Need State:

```
// Filter changes what projects are displayed
const [selectedCategory, setSelectedCategory] = useState('All')

<button onClick={() => setSelectedCategory('Frontend')}>
  Frontend
</button>
```

useState Syntax Explained:

```
const [count, setCount] = useState(0)
//     ↑       ↑              ↑
//  Current  Updater      Initial
//   value   function      value

// Read the value:
console.log(count)  // 0

// Update the value (triggers re-render):
setCount(5)  // Now count is 5

// Update based on previous value:
setCount(count + 1)  // Increment by 1
```

Complete Example - Counter:

```
function Counter() {
  // Declare state
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  )
}

// What happens when button is clicked:
// 1. onClick handler runs
```

```
// 2. setCount(count + 1) updates state
// 3. React re-renders the component
// 4. New count value appears on screen
```

Real Example from Your Portfolio:

```
// State for selected category
const [selectedCategory, setSelectedCategory] = useState('All')

// Filter projects based on state
const filteredProjects = selectedCategory === 'All'
  ? projects
  : projects.filter(p => p.category === selectedCategory)

// Button updates state
<button onClick={() => setSelectedCategory('Frontend')}>
  Frontend
</button>

// Display filtered results (changes when state changes!)
{filteredProjects.map(project => (
  <ProjectCard project={project} />
))}
```

Light Switch Analogy:

- **Without state**: Switch that doesn't control anything (static)
- **With state**: Switch that controls a light bulb (dynamic)

---

4. The .map() Function

What is .map()?

An array method that **transforms each item** and returns a new array.

Basic JavaScript Examples:

```
// Double numbers
const numbers = [1, 2, 3]
const doubled = numbers.map(num => num * 2)
// Result: [2, 4, 6]

// Uppercase names
const names = ["john", "jane"]
const uppercase = names.map(name => name.toUpperCase())
// Result: ["JOHN", "JANE"]

// Extract property from objects
```

```
const users = [
  { id: 1, name: "John" },
  { id: 2, name: "Jane" }
]
const names = users.map(user => user.name)
// Result: ["John", "Jane"]
```

In React - Creating Multiple Components:

```
const projects = [
  { id: 1, title: "Project A" },
  { id: 2, title: "Project B" },
  { id: 3, title: "Project C" }
]

// Transform array of data into array of components
{projects.map((project) => (
  <ProjectCard key={project.id} project={project} />
))}

// This creates:
<>
  <ProjectCard key={1} project={{ id: 1, title: "Project A" }} />
  <ProjectCard key={2} project={{ id: 2, title: "Project B" }} />
  <ProjectCard key={3} project={{ id: 3, title: "Project C" }} />
</>
```

Syntax Breakdown:

```
projects.map((project) => (
//  ↑          ↑       ↑  ↑
// Array    Each item | Return JSX
//                Arrow function

  <ProjectCard
    key={project.id}        // Unique identifier
    project={project}       // Pass entire object as prop
  />
))
```

Why the key prop?

React uses key to track which items changed, were added, or removed:

```
// ❌ BAD - React can't track items
{projects.map(project => (
  <ProjectCard project={project} />
```

```
))}

// ✅ GOOD - React knows each item uniquely
{projects.map(project => (
  <ProjectCard key={project.id} project={project} />
))}

// ❌ BAD - Using array index (problematic if list changes)
{projects.map((project, index) => (
  <ProjectCard key={index} project={project} />
))}

// ✅ GOOD - Use unique ID from data
{projects.map(project => (
  <ProjectCard key={project.id} project={project} />
))}
```

Real Example from Your CV:

```
// In Experience.tsx
{experience.description.map((point, idx) => (
  <li key={idx}>
    <span>•</span>
    <span>{point}</span>
  </li>
))}

// If experience.description = ["Built features", "Fixed bugs"]
// This creates:
<li key={0}>
  <span>•</span>
  <span>Built features</span>
</li>
<li key={1}>
  <span>•</span>
  <span>Fixed bugs</span>
</li>
```

---

5. Smooth Scrolling

What It Does:

Without smooth scrolling:

```
Click link → Page JUMPS instantly to section
```

Jarring, teleportation-like

With smooth scrolling:

```
Click link → Page GLIDES smoothly to section
```

Animated, professional feel

How We Enabled It:

```
/* In app/globals.css */
html {
  scroll-behavior: smooth;
}
```

This **single CSS line** makes ALL anchor link navigation smooth!

How It Works:

```
<!-- Navbar link -->
<a href="#portfolio">Portfolio</a>

<!-- Target section -->
<section id="portfolio">...</section>
```

Without smooth scrolling: Instant jump **With smooth scrolling:** Smooth animated scroll

Browser Support:

- ✅ Chrome, Firefox, Edge, Safari (modern versions)
- For older browsers, you'd need JavaScript polyfill

---

6. Tailwind CSS

Philosophy:

Instead of writing CSS files, apply **utility classes** directly in HTML/JSX.

Traditional CSS vs Tailwind:

Traditional Approach:

```
/* styles.css */
.hero-section {
  display: flex;
  flex-direction: column;
```

```
  align-items: center;
  gap: 16px;
  padding: 32px;
  background-color: #f0f0f0;
  border-radius: 8px;
}
```

```
<div class="hero-section">Content</div>
```

Tailwind Approach:

```
<!-- Everything in one place! -->
<div className="flex flex-col items-center gap-4 p-8 bg-gray-100 rounded-lg">
  Content
</div>
```

Common Classes Reference:

Tailwind Class	CSS Equivalent	What It Does
`flex`	`display: flex`	Flexbox container
`flex-col`	`flex-direction: column`	Stack vertically
`flex-row`	`flex-direction: row`	Arrange horizontally
`items-center`	`align-items: center`	Center items on cross axis
`justify-center`	`justify-content: center`	Center items on main axis
`gap-4`	`gap: 1rem` (16px)	Space between items
`p-4`	`padding: 1rem`	Padding all sides
`px-4`	`padding-left/right: 1rem`	Horizontal padding only
`py-4`	`padding-top/bottom: 1rem`	Vertical padding only
`m-4`	`margin: 1rem`	Margin all sides
`w-full`	`width: 100%`	Full width
`h-screen`	`height: 100vh`	Full viewport height
`bg-blue-500`	`background-color: #3b82f6`	Blue background
`text-white`	`color: white`	White text
`font-bold`	`font-weight: bold`	Bold text
`text-2xl`	`font-size: 1.5rem`	Large text

Tailwind Class	CSS Equivalent	What It Does
`rounded-lg`	`border-radius: 0.5rem`	Rounded corners
`border`	`border: 1px solid`	Add border

Spacing Scale (Important!):

Tailwind uses multiples of 4px:

```
gap-1  = 4px    (0.25rem)
gap-2  = 8px    (0.5rem)
gap-3  = 12px   (0.75rem)
gap-4  = 16px   (1rem)
gap-6  = 24px   (1.5rem)
gap-8  = 32px   (2rem)
gap-12 = 48px   (3rem)
gap-16 = 64px   (4rem)
```

Responsive Design:

```
<!-- Traditional CSS -->
<style>
  .container { flex-direction: column; }

  @media (min-width: 768px) {
    .container { flex-direction: row; }
  }
</style>
<div class="container">...</div>

<!-- Tailwind (much simpler!) -->
<div className="flex-col md:flex-row">...</div>
<!--     ↑ default    ↑ medium screens and up -->
```

Breakpoint prefixes:

- (none) = all screen sizes
- `sm:` = 640px and up (small tablets)
- `md:` = 768px and up (tablets)
- `lg:` = 1024px and up (laptops)
- `xl:` = 1280px and up (desktops)

Real Example from Your Hero:

```
<div className="flex flex-col md:flex-row items-center gap-12">
//              ↓        ↓          ↓           ↓          ↓
```

```
//          Display  Column on   Row on      Center    48px
//           flex    mobile     desktop      items     gap

// What this means:
// - Always: display: flex, align-items: center, gap: 48px
// - Mobile (<768px): flex-direction: column
// - Desktop (≥768px): flex-direction: row
```

Dark Mode with Custom Colors:

Your CV uses CSS variables for automatic dark mode:

```
/* globals.css */
:root {
  --background: #ffffff;  /* Light mode */
  --foreground: #171717;
}

@media (prefers-color-scheme: dark) {
  :root {
    --background: #0a0a0a;  /* Dark mode */
    --foreground: #ededed;
  }
}

@theme inline {
  --color-background: var(--background);
  --color-foreground: var(--foreground);
}
```

```
<!-- Automatically adapts to system theme! -->
<div className="bg-background text-foreground">
  Light in light mode, dark in dark mode
</div>
```

Pseudo-classes (Hover, Focus, etc.):

```
<!-- Traditional CSS -->
<style>
  .button { background: blue; }
  .button:hover { background: darkblue; }
</style>

<!-- Tailwind -->
<button className="bg-blue-500 hover:bg-blue-700">
  <!--                    ↑ Only applies on hover -->
```

```
  Click me
</button>
```

Other pseudo-classes:

- `hover:` - On mouse hover
- `focus:` - When input is focused
- `active:` - When button is pressed
- `disabled:` - When disabled
- `group-hover:` - Hover on parent affects child

Example: Hover Effect on Card

```
<div className="bg-white hover:bg-gray-100 transition-colors">
  <!--                          ↑ Changes background on hover
                                    ↑ Smooth color transition
  -->
  Card content
</div>
```

---

7. Key Takeaways

React Mental Model:

1. **Components** = Functions that return UI
2. **Props** = Arguments passed to components
3. **State** = Data that can change (triggers re-render)
4. **Rendering** = React updates the DOM when data changes

Component Hierarchy:

```
App (layout.tsx)
└── Page (page.tsx)
    ├── Navigation
    ├── Hero
    ├── Experience
    │   └── ExperienceCard (multiple instances)
    ├── Skills
    ├── Portfolio
    │   └── ProjectCard (multiple instances)
    ├── Contact
    └── Footer
```

Data Flow:

```
lib/data.ts (source of truth)
    ↓
Component imports data
    ↓
Component passes data as props
    ↓
Child component receives props
    ↓
Child renders UI with data
```

When to Use What:

Regular HTML/Links:

- Navigation
- External links
- Static content

State (useState):

- Filters/sorting
- Form inputs
- Toggle switches
- Counters
- Any data that changes based on user interaction

Props:

- Passing data from parent to child
- Configuration for reusable components
- Event handlers

React vs. Server-Side Rendering (EJS):

Your MealCreator (EJS):

```
User clicks → Server renders new HTML → Full page reload
```

This CV (React/Next.js):

```
Initial load → React takes over → Updates only what changed
```

Benefits:

- Faster interactions (no page reload)
- Smoother user experience

- Better for dynamic content
- Next.js adds SEO benefits through server rendering

Quick Reference Commands

```
# Development
npm run dev          # Start dev server (http://localhost:3000)
npm run build        # Build for production
npm start            # Run production build

# File structure
/app                 # Pages and layouts
/components          # Reusable UI components
/lib                 # Data and utilities
/public              # Static assets (images, etc.)
```

Learning Resources

Official Docs:

- React: https://react.dev
- Next.js: https://nextjs.org/docs
- Tailwind CSS: https://tailwindcss.com/docs

Key Concepts to Explore Next:

1. **useEffect** - Side effects (API calls, subscriptions)
2. **Custom Hooks** - Reusable logic
3. **Context API** - Global state management
4. **Server vs Client Components** - Next.js App Router feature
5. **Data Fetching** - How to load data in Next.js

Questions to Test Understanding

1. What's the difference between props and state?
2. When would you use .map() in React?
3. Why do we need a `key` prop when mapping?
4. What is JSX and how is it different from JSON?
5. When does a component re-render?
6. What's the benefit of Tailwind over traditional CSS?
7. What does `md:flex-row` mean in Tailwind?

Answers:

1. **Props** are passed from parent (read-only). **State** is internal to component (can change).

2. Use .map() to **render multiple items** from an array (e.g., list of projects).
3. `key` helps React **track which items changed** for efficient updates.
4. **JSX** is HTML-like syntax in JavaScript. **JSON** is a data format. JSX gets compiled to JavaScript, JSON is for data transfer.
5. Component re-renders when:
    - Its state changes (useState)
    - Its props change
    - Parent component re-renders
6. **Tailwind**: Faster development, no naming classes, consistent design, easy responsive design.
   Traditional CSS: More control, separation of concerns, familiar to many developers.
7. `md:flex-row` = "On **medium screens and up** (≥768px), apply `flex-row`"

Your Project Structure

```
CV/
├── app/
│   ├── globals.css          # Global styles + CSS variables
│   ├── layout.tsx           # Root layout (wraps entire app)
│   ├── page.tsx             # Home page (assembles all sections)
│   └── favicon.ico          # Site icon
├── components/
│   ├── Navigation.tsx       # Fixed navbar
│   ├── Hero.tsx             # Hero section with photo & intro
│   ├── Experience.tsx       # Work history timeline
│   ├── Skills.tsx           # Skills by category
│   ├── Portfolio.tsx        # Project showcase with filtering ☆ Uses state!
│   ├── Contact.tsx          # Contact info & social links
│   └── Footer.tsx           # Page footer
├── lib/
│   ├── data.ts              # ALL CV content (easy to edit!)
│   └── types.ts             # TypeScript type definitions
├── public/
│   ├── images/              # Project images & profile photo
│   └── resume.pdf           # Downloadable CV (add this!)
└── package.json             # Project dependencies
```

STAGE 2 - ADVANCED INTERACTIONS & ANIMATIONS

---

8. useState Hook

What Is It?

Component memory - lets React components remember values that change over time.

Syntax:

```
const [value, setValue] = useState(initialValue);
//     ↑       ↑              ↑
//   Current  Updater      Starting
//   value    function     value
```

When to Use:

- Theme preference (dark/light mode)
- Menu open/closed state
- Active section tracking
- Filter selections
- Form inputs
- Any data that changes based on user interaction

Key Rules:

1. Call at **top of component** only (not in loops/conditions)
2. **Never mutate state directly** - always use setter function
3. State updates are **asynchronous**

Example from Your CV:

```
// Theme toggle
const [isDark, setIsDark] = useState(false);

// User clicks → updates state → React re-renders → UI updates
<button onClick={() => setIsDark(!isDark)}>
  {isDark ? <MoonIcon /> : <SunIcon />}
</button>
```

---

9. useEffect Hook

What Is It?

Runs code **after** component renders (for side effects like API calls, event listeners).

Syntax:

```
useEffect(() => {
  // Code runs after render
```

```
  return () => {
    // Cleanup (optional)
  };
}, [dependencies]); // When to re-run
```

Dependency Array:

- `[]` = Run **once** on mount
- `[count]` = Re-run when `count` changes
- No array = Run on **every** render (usually wrong!)

When to Use:

- Load saved preferences from localStorage
- Add/remove event listeners
- Fetch data from APIs
- Track scroll position
- Update document title

Example from Your CV:

```
// Load theme on mount
useEffect(() => {
  const savedTheme = localStorage.getItem('theme');
  if (savedTheme === 'dark') {
    setIsDark(true);
  }
}, []); // Empty array = run once

// Track scroll position
useEffect(() => {
  const handleScroll = () => {
    setScrollProgress(window.scrollY);
  };

  window.addEventListener('scroll', handleScroll);

  // CLEANUP: Remove listener on unmount
  return () => window.removeEventListener('scroll', handleScroll);
}, []);
```

---

10. localStorage API

What Is It?

Browser storage that persists data even after closing the page.

API:

```
localStorage.setItem('key', 'value');     // Save
const value = localStorage.getItem('key'); // Load (returns string or null)
localStorage.removeItem('key');            // Delete
localStorage.clear();                      // Delete all
```

Important:

- Only stores **strings**
- For objects, use `JSON.stringify()` and `JSON.parse()`
- Check for `null` (key might not exist)
- ~5-10MB storage limit

Example from Your CV:

```
// Save theme preference
localStorage.setItem('theme', 'dark');

// Load theme preference
const savedTheme = localStorage.getItem('theme');
const isDark = savedTheme === 'dark';

// Store complex data
const prefs = { theme: 'dark', fontSize: 'large' };
localStorage.setItem('prefs', JSON.stringify(prefs));
const loaded = JSON.parse(localStorage.getItem('prefs'));
```

---

11. Framer Motion Animations

What Is It?

Animation library for React - easier than CSS keyframes.

Basic Syntax:

```
<motion.div
  initial={{ opacity: 0, y: 40 }}      // Start state
  animate={{ opacity: 1, y: 0 }}       // End state
  transition={{ duration: 0.7 }}       // How long
>
  Content
</motion.div>
```

Common Props:

- `initial` - Starting state (before animation)

- `animate` - Ending state (what it animates to)
- `transition` - Speed, easing, delay
- `whileHover` - Style while hovering
- `exit` - State when leaving (needs AnimatePresence)

useInView Hook:

```
const ref = useRef(null);
const isInView = useInView(ref, { once: true });

<motion.div
  ref={ref}
  animate={{ opacity: isInView ? 1 : 0 }}
>
```

Example from Your CV:

```
// ScrollReveal component
<ScrollReveal delay={0.2}>
  <div>Fades in on scroll</div>
</ScrollReveal>

// Hero image scale-in
<motion.div
  initial={{ opacity: 0, scale: 0.5 }}
  animate={{ opacity: 1, scale: 1 }}
>
  <Image src="/profile.jpg" />
</motion.div>
```

---

12. Event Listeners

What Are They?

Functions that respond to browser events (scroll, click, resize, etc.)

Adding Listeners:

```
window.addEventListener('scroll', handleScroll);
window.addEventListener('resize', handleResize);
```

Removing Listeners (Important!):

```
window.removeEventListener('scroll', handleScroll);
```

Always remove listeners in useEffect cleanup to prevent memory leaks!

Example from Your CV:

```
useEffect(() => {
  const handleScroll = () => {
    // Calculate scroll progress
    const progress = (window.scrollY / totalHeight) * 100;
    setScrollProgress(progress);

    // Check which section is visible
    const section = document.getElementById('experience');
    const rect = section.getBoundingClientRect();
    if (rect.top <= 100) {
      setActiveSection('experience');
    }
  };

  window.addEventListener('scroll', handleScroll);

  // Cleanup on unmount
  return () => window.removeEventListener('scroll', handleScroll);
}, []);
```

---

13. Stage 2 Summary

Features Built:

✅ **Smooth scroll animations** - ScrollReveal component with Framer Motion
✅ **Active section tracking** - Nav highlights current section
✅ **Mobile menu** - Hamburger menu with animation
✅ **Scroll progress bar** - Shows reading progress
✅ **Theme toggle** - Dark/light mode with persistence
✅ **Micro-interactions** - Hover effects on cards and buttons

Technologies Learned:

- **useState** - Component memory for changing data
- **useEffect** - Side effects after render
- **localStorage** - Persist data in browser
- **Framer Motion** - Smooth animations
- **Event Listeners** - Respond to scroll, click, etc.
- **useRef** - Reference DOM elements
- **CSS Transforms** - GPU-accelerated animations

Key Patterns:

```
// 1. Track state
const [value, setValue] = useState(initial);

// 2. Load saved data after mount
useEffect(() => {
  const saved = localStorage.getItem('key');
  setValue(saved);
}, []);

// 3. Save on change
useEffect(() => {
  localStorage.setItem('key', value);
}, [value]);

// 4. Cleanup listeners
useEffect(() => {
  window.addEventListener('event', handler);
  return () => window.removeEventListener('event', handler);
}, []);
```

Questions to Test Understanding (Stage 2)

1. What's the difference between useState and regular variables?
2. When does useEffect run?
3. Why do we need cleanup functions in useEffect?
4. What does localStorage.getItem() return if the key doesn't exist?
5. How do you store an object in localStorage?
6. What's the difference between `initial` and `animate` in Framer Motion?
7. Why use CSS transforms instead of changing width/height?

Answers:

1. **useState** triggers re-render when changed. Regular variables don't trigger re-renders.
2. useEffect runs **after** the component renders. Dependency array controls when it re-runs.
3. Cleanup prevents **memory leaks** by removing event listeners and canceling subscriptions when component unmounts.
4. Returns **null** (not undefined). Always check: `const value = localStorage.getItem('key') || 'default'`
5. Convert to JSON: `localStorage.setItem('user', JSON.stringify(userObject))` then parse: `JSON.parse(localStorage.getItem('user'))`
6. **initial** = starting state (invisible). **animate** = ending state (visible). Framer Motion animates between them.
7. Transforms use **GPU** (faster, smoother). Width/height changes trigger layout recalculation (slow, janky).

This document created: October 19, 2025 Updated with Stage 2: October 19, 2025 Updated with Stage 3: October 19, 2025 Project: Interactive CV Website Tech Stack: Next.js 15, React 19, TypeScript, Tailwind CSS v4, Framer Motion

STAGE 3 - VIDEO INTEGRATION & MEDIA HANDLING

14. Video Hover System

What We Built

A sophisticated hover system that **replaces static images with videos** when hovering over portfolio project cards.

The Challenge

We needed to:

1. Show a video when hovering over a project card
2. Play the video **only once** (not loop)
3. Show the image again after video ends
4. Reset video when user leaves and returns
5. Make videos responsive and fit properly

Type System Extension

First, we extended the `Project` type to support optional videos:

```
// lib/types.ts
export interface Project {
  id: number;
  title: string;
  // ... other fields
  image: string;
  video?: string;  // ← NEW: Optional video URL
  // ... other fields
}
```

Why optional (`?`)? Not all projects have videos. This makes it backwards compatible.

State Management Strategy

```
// components/Portfolio.tsx - ProjectCard component
const [isHovering, setIsHovering] = useState(false);
const [videoEnded, setVideoEnded] = useState(false);
```

Two separate states - why?

1. `isHovering` - Tracks mouse position (over card or not)
2. `videoEnded` - Tracks if video finished playing

The Logic Flow:

```
User hovers → isHovering = true → Video plays
Video finishes → videoEnded = true → Show image
User still hovering → Image stays (video ended)
User leaves → Both states reset
User returns → Video plays again from start
```

Event Handler Pattern

```
const handleMouseEnter = () => {
  setIsHovering(true);
  setVideoEnded(false); // Reset video state
};

const handleMouseLeave = () => {
  setIsHovering(false);
  setVideoEnded(false); // Reset for next visit
};
```

Why reset `videoEnded` on both enter AND leave?

- On **enter**: Prepare for new video playback
- On **leave**: Clean slate for next hover

Conditional Rendering Logic

```
{isHovering && project.video && !videoEnded ? (
  <video src={project.video} ... />
) : (
  <Image src={project.image} ... />
)}
```

Breaking down the condition:

```
isHovering          // User's mouse is over the card
&&                  // AND
project.video       // Project has a video URL
&&                  // AND
!videoEnded         // Video hasn't finished yet
```

```
? <video />          // Then show video
: <Image />          // Otherwise show image
```

Video Element Properties

```
<video
  src={project.video}
  autoPlay           // Start immediately
  muted              // Required for autoPlay in browsers
  playsInline        // Prevent fullscreen on mobile
  onEnded={() => setVideoEnded(true)}  // Track when finished
  className="absolute inset-0 w-full h-full object-contain"
/>
```

Key Properties Explained:

Property	Purpose	Why?
`autoPlay`	Starts playing immediately	No click needed
`muted`	Silences audio	Browsers block unmuted autoplay
`playsInline`	Plays in element (not fullscreen)	Better mobile UX
`onEnded`	Event when video finishes	Triggers state update

No `loop` attribute - We deliberately excluded this so video plays once.

CSS Object Fit - Making Videos Responsive

```
className="absolute inset-0 w-full h-full object-contain"
```

CSS Breakdown:

- `absolute inset-0` = Position in top-left, stretch to container edges
- `w-full h-full` = Take full width and height
- `object-contain` = Fit entire video without cropping

object-contain vs object-cover:

```
object-contain:
┌─────────────────┐
│   ┌───────┐     │  ← Entire video visible
│   │VIDEO  │     │    Black bars may appear
│   └───────┘     │    Maintains aspect ratio
└─────────────────┘
```

```
object-cover:
┌──────────────┐
│▓▓▓▓VIDEO▓▓▓▓│  ← Fills container
│▓▓▓▓VIDEO▓▓▓▓│    Parts may be cropped
│▓▓▓▓VIDEO▓▓▓▓│    No gaps
└──────────────┘
```

We used `object-contain` because project demos need to be fully visible.

Hover Target Expansion

Initially, video played only when hovering over the image. We moved the event handlers to the entire card:

```
<div
  className="..."
  onMouseEnter={handleMouseEnter}  // ← On card, not just image
  onMouseLeave={handleMouseLeave}
>
  <div className="image-container">...</div>
  <div className="project-info">...</div>  // ← Hovering here also triggers video
</div>
```

Why? Better UX - video stays playing while reading project details.

Per-Project Image Styling

Different projects needed different image treatments:

```
className={`... ${
  project.title.includes('MealCreator')
    ? 'object-cover'              // Fill container, may crop
  : project.title.includes('TheraBot')
    ? 'object-contain'            // Show entire image
  : project.title.includes('WhatsApp')
    ? 'object-cover object-[center_90%]'  // Cover, offset to 90%
  : project.title.includes('Interactive CV')
    ? 'object-contain'            // Show entire image
  : 'object-contain'              // Default
}`}
```

Custom Background Colors:

```
className={`relative h-48 overflow-hidden ${
  project.title.includes('TheraBot')
    ? 'bg-[#fcf7ed]'              // Cream background
  : project.title.includes('WhatsApp')
    ? 'bg-[#0c0f12]'              // Dark background
```

```
    : project.title.includes('Interactive CV')
      ? 'bg-black'                  // Black background
    : 'bg-foreground/10'            // Default semi-transparent
  }`}
```

Why different backgrounds? Some images have transparent areas or specific color schemes.

---

15. HTML Video Element Deep Dive

Video vs Image in React

Image Component (Next.js):

```
<Image src="/image.png" fill alt="..." />
```

- Automatic optimization
- Lazy loading
- Responsive images
- `fill` prop for container-based sizing

Video Element:

```
<video src="/video.mp4" autoPlay muted playsInline />
```

- Native HTML element
- No automatic optimization
- Need manual responsive handling
- More control over playback

Video File Considerations

Video Formats:

- **MP4 (H.264)** - Best browser support, good compression
- **WebM** - Better compression, less browser support
- **OGG** - Open source, declining support

Best Practice:

```
<video>
  <source src="/video.mp4" type="video/mp4" />
  <source src="/video.webm" type="video/webm" />
  Your browser doesn't support video.
</video>
```

Performance Tips:

- Keep videos short (10-30 seconds)
- Compress before uploading
- Consider video hosting (Vimeo, YouTube) for longer videos
- Use appropriate resolution (1080p max for web)

Browser Autoplay Policies

Modern browsers block autoplay videos with sound to prevent annoying users.

The Rule:

```
Autoplay WITHOUT sound = ✅ Allowed
Autoplay WITH sound = ❌ Blocked (unless user interacted with site)
```

That's why we need:

```
<video autoPlay muted>  // muted is REQUIRED for autoPlay
```

Video Events

The `<video>` element fires many events we can listen to:

```
<video
  onPlay={() => console.log('Started')}
  onPause={() => console.log('Paused')}
  onEnded={() => console.log('Finished')}  // ← We use this one
  onTimeUpdate={(e) => console.log(e.currentTarget.currentTime)}
  onLoadedData={() => console.log('Video loaded')}
  onError={() => console.log('Failed to load')}
/>
```

Common Use Cases:

- `onEnded` - Show different UI when video finishes
- `onTimeUpdate` - Build custom progress bar
- `onError` - Show fallback image if video fails

---

16. Advanced State Patterns

Multiple Related States

```
const [isHovering, setIsHovering] = useState(false);
const [videoEnded, setVideoEnded] = useState(false);
```

When to use multiple states vs one object?

```
// ❌ Could do this (but more complex):
const [videoState, setVideoState] = useState({
  isHovering: false,
  videoEnded: false
});

// ✅ Better - separate states for unrelated concerns:
const [isHovering, setIsHovering] = useState(false);
const [videoEnded, setVideoEnded] = useState(false);
```

Rule of thumb:

- **Separate states** if they change independently
- **Object state** if values are always updated together

State Reset Pattern

```
const handleMouseEnter = () => {
  setIsHovering(true);
  setVideoEnded(false);  // Reset dependent state
};
```

Key Pattern: When one state changes, reset related states that depend on it.

Derived State (Don't Overuse State!)

```
// ❌ BAD - Unnecessary state
const [showVideo, setShowVideo] = useState(false);
useEffect(() => {
  setShowVideo(isHovering && !videoEnded);
}, [isHovering, videoEnded]);

// ✅ GOOD - Calculate directly
const showVideo = isHovering && project.video && !videoEnded;
```

Rule: If you can calculate a value from existing state, don't store it in separate state.

---

17. Component Composition Patterns

Before: Simple Component

```
function ProjectCard({ project }) {
  return (
    <div>
      <Image src={project.image} />
      <h3>{project.title}</h3>
    </div>
  );
}
```

After: Complex Interactive Component

```
function ProjectCard({ project }) {
  // State management
  const [isHovering, setIsHovering] = useState(false);
  const [videoEnded, setVideoEnded] = useState(false);

  // Event handlers
  const handleMouseEnter = () => {
    setIsHovering(true);
    setVideoEnded(false);
  };

  const handleMouseLeave = () => {
    setIsHovering(false);
    setVideoEnded(false);
  };

  // Conditional rendering logic
  return (
    <div onMouseEnter={handleMouseEnter} onMouseLeave={handleMouseLeave}>
      {isHovering && project.video && !videoEnded ? (
        <video ... />
      ) : (
        <Image ... />
      )}
      <h3>{project.title}</h3>
    </div>
  );
}
```

Component Evolution:

1. **Stage 1**: Static presentation
2. **Stage 2**: Basic interactivity (hover effects)
3. **Stage 3**: Complex state-driven behavior (video system)

---

18. Data Structure Evolution

Type System Growth

```
// Stage 1: Basic project data
interface Project {
  id: number;
  title: string;
  image: string;
}

// Stage 2: Added optional fields
interface Project {
  id: number;
  title: string;
  image: string;
  liveUrl?: string;  // Optional
  githubUrl?: string; // Optional
}

// Stage 3: Added video support
interface Project {
  id: number;
  title: string;
  image: string;
  video?: string;  // NEW: Optional video
  liveUrl?: string;
  githubUrl?: string;
}
```

Progressive Enhancement: Each stage adds features without breaking existing functionality.

Data File Organization

```
// lib/data.ts
export const projects = [
  {
    id: 1,
    title: "MealCreator Website",
    image: "/images/mealcreator.png",
    video: "/videos/mealcreator.mp4",  // ← Added in Stage 3
    status: "In Development",           // ← Status badge
    // ... rest of data
  },
];
```

Separation of Concerns:

- **lib/types.ts** - Type definitions (structure)

- **lib/data.ts** - Actual content (data)
- **components/** - Presentation logic (UI)

This makes it easy to:

- Update content without touching code
- Change types without touching data
- Modify UI without touching content

---

19. Stage 3 Summary

Features Built

✅ **Video hover system** - Play project demo videos on hover
✅ **Smart video playback** - Plays once, returns to image
✅ **Responsive video sizing** - Fits perfectly in containers
✅ **Per-project styling** - Custom backgrounds and object-fit
✅ **Enhanced hover targets** - Video plays when hovering anywhere on card

Technical Concepts Learned

React Patterns:

- Multiple coordinated state variables
- State reset patterns
- Derived state vs stored state
- Complex conditional rendering

HTML5 Video:

- Video element properties and methods
- Browser autoplay policies
- Video events (onEnded)
- Responsive video sizing with CSS

CSS Techniques:

- `object-fit` (contain vs cover)
- `object-position` for fine-tuning
- Absolute positioning with `inset-0`
- Custom background colors per component

TypeScript:

- Optional properties (`?`)
- Type system evolution
- Progressive enhancement

Code Patterns Summary

```
// Pattern 1: Multiple States for Complex Interactions
const [isHovering, setIsHovering] = useState(false);
const [videoEnded, setVideoEnded] = useState(false);

// Pattern 2: Coordinated State Updates
const handleMouseEnter = () => {
  setIsHovering(true);
  setVideoEnded(false); // Reset related state
};

// Pattern 3: Complex Conditional Rendering
{condition1 && condition2 && !condition3 ? (
  <ComponentA />
) : (
  <ComponentB />
)}

// Pattern 4: Event Callbacks That Update State
<video onEnded={() => setVideoEnded(true)} />

// Pattern 5: Responsive Media with Object-Fit
<video className="absolute inset-0 w-full h-full object-contain" />
```

Real-World Applications

These patterns are used in:

- **Netflix/YouTube** - Video players with complex state
- **Instagram/TikTok** - Auto-playing content on scroll
- **E-commerce** - Product image/video galleries
- **Landing pages** - Hero videos with fallback images
- **Documentation** - Animated examples

---

20. Questions to Test Understanding (Stage 3)

1. Why do we need TWO state variables (isHovering and videoEnded)?
2. What happens if you remove `muted` from a video with `autoPlay`?
3. Why reset `videoEnded` in both mouseEnter AND mouseLeave?
4. What's the difference between `object-contain` and `object-cover`?
5. Why did we remove the `loop` attribute from the video?
6. When should you use derived state vs useState?
7. Why is the video positioned with `absolute inset-0`?

Answers:

1. **Two states needed** because they track different things: one tracks mouse position (hover), one tracks video completion. They change at different times.

2. **Browser blocks autoplay** - Most browsers prevent videos with sound from autoplaying to avoid annoying users. Video won't play without user interaction.
3. **Reset on enter** = Prepare for new playback. **Reset on leave** = Clean state for next visit. This ensures video always restarts from beginning.
4. **object-contain** = Shows entire content, may have empty space. **object-cover** = Fills container completely, may crop content.
5. **UX decision** - Playing once is less distracting. User can see demo, then it returns to static state. If they want to see again, they can hover again.
6. **Use derived state** when value can be calculated from existing state/props. **Use useState** when you can't calculate it (like user input, API response, time-based changes).
7. **absolute inset-0** positions the video to fill its container completely. It's positioned relative to the parent and stretches to all edges (top: 0, right: 0, bottom: 0, left: 0).

---

21. Performance Considerations

Video Loading

Videos are **much larger** than images:

- Image: 50-500 KB
- Video: 1-10 MB or more

Optimization strategies:

1. **Lazy loading** - Only load videos when needed
2. **Compression** - Reduce file size
3. **Resolution** - Don't use 4K for small displays
4. **Streaming** - Use CDN or video hosting service

Current Implementation

```
<video src={project.video} />
```

This loads the video immediately when hovering. For production, consider:

```
// Preload just metadata, not full video
<video src={project.video} preload="metadata" />

// Or lazy load completely
{isHovering && (
  <video src={project.video} />
)}
```

React Re-renders

Our implementation is efficient because:

1. **State is local** to each ProjectCard (not global)
2. **Only the hovered card re-renders** (not all cards)
3. **Video starts/stops immediately** (no loading delay from state updates)

Congratulations! 🎉

Stage 3 Complete! You now understand:

- Complex state coordination
- HTML5 video element
- Media optimization
- Advanced conditional rendering
- Event-driven UI updates
- CSS object-fit for responsive media

You've built a production-ready portfolio feature that showcases your projects with professional video demos!

Next Learning Steps

Stage 3 Complete! ✅ You now have:

- Advanced React state patterns
- Media handling expertise
- Complex user interaction flows

Possible Next Steps:

1. **Performance optimization**: Lazy loading, code splitting
2. **Custom Hooks**: Extract reusable video logic
3. **Context API**: Global state without prop drilling
4. **Testing**: Jest and React Testing Library
5. **Accessibility**: Keyboard navigation, ARIA labels
6. **Deployment**: Deploy to Vercel/Netlify
7. **Analytics**: Track user interactions

Remember: **You learn by building.** This CV project is giving you real-world React experience!