The Comprehensive Manual of Atomic Architecture in React & Vite Engineering

Executive Summary and Architectural Vision

Welcome to the engineering team. You are entering a codebase that has evolved beyond the simple aggregation of scripts and styles; you are stepping into a structured ecosystem designed for scalability, maintainability, and cognitive clarity. In the modern landscape of frontend engineering, particularly within the React and Vite ecosystem, the primary challenge is not merely mastering syntax—it is mastering the **mental model** of the application.

This document serves as the definitive guide for navigating, contributing to, and mastering the **Atomic Design Architecture** that governs our project. We utilize this methodology not because it is trendy, but because it solves the fundamental problem of UI development: entropy. As applications grow, they tend to become chaotic. Components become entangled, styles leak, and business logic scatters. Atomic Design provides a rigid yet flexible hierarchy—derived from the natural world—to prevent this chaos.[1]

This report is structured to take you from a complete novice understanding—assuming zero prior knowledge of this specific architecture—to a professional level of proficiency. We will dissect the directory structure, the theoretical underpinnings of component isolation, the nuances of atomic state management (using Jotai/Recoil principles), and the "Pro Level" workflows involving Storybook, Vitest, and strict dependency enforcement. This is not just a tutorial; it is the operational doctrine of our frontend engineering culture.

Part I: The Theoretical Foundation of Atomic Systems

1.1 The Failure of Monolithic Design

To understand why we use Atomic Design, one must first understand the problem it eradicates. In early web development, and even in poorly structured React applications, interfaces are often built as "Pages." A developer is tasked with building a "Dashboard," so they create a Dashboard.tsx file and write 2,000 lines of code including the header, the charts, the user profile, and the footer.

This "Monolithic Page" approach creates several critical failures:

1. **Duplication:** When the "User Profile" needs to appear on the "Settings Page," the code is often copy-pasted, leading to drift. If the brand color changes, you must update it in fifty places.
2. **Cognitive Load:** For a new intern, opening a 2,000-line file is paralyzing. It is impossible

to isolate where the "Submit Button" logic lives amidst the layout code.
3. **Untestability:** You cannot unit test a "Dashboard." You can only integration test it. If the button fails, does the whole dashboard fail?

1.2 The Chemistry Analogy: A Mental Model for UIs

Atomic Design, pioneered by Brad Frost, asks us to stop looking at web pages as "pages" and start looking at them as systems of components.[1] The methodology draws a direct parallel to chemistry. In the natural world, all matter is composed of **atoms**. Atoms bond together to form **molecules**. Molecules combine to form complex **organisms**.

In our React project, this analogy is literal. Our file system reflects this hierarchy. We do not design a "Navigation Bar" from scratch; we assemble it from existing molecules (Links, Search Bars, Icons), which are themselves assembled from atoms (Text, Input, SVG).[1]

This hierarchy creates a "Vocabulary" for our UI. When you join the project, you don't need to memorize every file. You only need to understand the *category* of a component to know its complexity, its allowed dependencies, and its testing strategy.

1.3 The Five Distinct Stages of Hierarchy

The architecture is strictly divided into five stages. Understanding the boundaries between these stages is the most critical skill you will learn. A violation of these boundaries (e.g., an Atom importing an Organism) is a structural defect that will be flagged during code review.[3]

1.3.1 Atoms: The Indivisible Building Blocks

Atoms are the foundational elements of our user interface. By definition, an atom cannot be broken down any further without ceasing to be functional. In React terms, these are our lowest-level components.

- **Scope:** Atoms define the visual language of the application. They include global styles (fonts, colors, animations) and basic HTML elements wrapped in React logic.
- **Examples:** <Button>, <Input>, <Label>, <Icon>, <Avatar>, <Spinner>, <Typography>.
- **The "Pro" Nuance:** An intern might think an Atom is just an HTML tag. A Pro knows that an Atom handles **all** the edge cases of that HTML tag. A <Button> atom in our system handles the loading state, the disabled state, the focus state for accessibility, and the variant styling (Primary vs. Secondary) via props. It creates a robust API for the rest of the app to consume.
- **Rule:** Atoms never import Molecules or Organisms. They only import logic from utils or styles/tokens.[5]

1.3.2 Molecules: The Functional Groups

Molecules are relatively simple groups of UI elements working together as a unit. They give *purpose* to Atoms. An <Input> atom is abstract; it could be anything. A <Label> atom is just text. But when you combine a <Label>, an <Input>, and a <Button>, you create a SearchForm molecule.

- **Scope:** Molecules handle "local" functionality. They define the layout of the atoms they contain.
- **Examples:** SearchBar (Input + Button), UserCard (Avatar + Name Label), FormField (Label + Input + ErrorText), PaginationControls.
- **The "Pro" Nuance:** Molecules should be flexible enough to be reused in different contexts but specific enough to be useful. A UserCard molecule shouldn't care *which* user it is displaying; it just accepts name and avatarUrl as props.[7]

1.3.3 Organisms: The Distinct Interface Sections

Organisms are relatively complex UI components composed of groups of molecules and/or atoms and/or other organisms. These form distinct sections of an interface. This is often where the "business domain" begins to permeate the UI.

- **Scope:** Organisms typically define a standalone section of a page.
- **Examples:** Header (Logo Atom + Nav Molecules + Search Molecule), ProductList (Grid of ProductCard Molecules), Footer, Sidebar.
- **The "Pro" Nuance:** Organisms are often responsible for layout and grid positioning of the molecules within them. While molecules say "I look like this," Organisms say "I arrange these things like this".[1]

1.3.4 Templates: The Content Structure

Templates are page-level objects that place components into a layout and articulate the design's underlying content structure. Crucially, **Templates do not hold data.** They are skeletal structures.

- **Scope:** Templates define the grid, the responsive behavior, and the regions (e.g., "Left Sidebar," "Main Content," "Right Rail").
- **Examples:** DashboardTemplate, AuthTemplate (Centered card), BlogTemplate (Header + Article + Sidebar + Footer).
- **The "Pro" Nuance:** Templates allow us to change the layout of 50 pages by editing one file. If we want to move the sidebar from left to right, we update the DashboardTemplate, and every page using it updates instantly.[1]

1.3.5 Pages: The Context Instances

Pages are specific instances of templates. This is where the application comes alive. Pages are responsible for connecting the "Dumb" UI (Templates) with the "Smart" Data (API calls, Global State, Routing).

- **Scope:** High-level orchestration.
- **Examples:** HomePage, UserProfilePage, LoginPage.
- **The "Pro" Nuance:** Pages should contain very little UI logic. Their primary job is fetching

data (using React Query or standard hooks) and passing it down to the Template.[8]

Part II: The React & Vite Engineering Ecosystem

2.1 Why Vite? The Modern Build Tool

In the past, Create React App (CRA) was the standard. However, as our project scales, CRA's Webpack-based architecture becomes sluggish. We use **Vite**.[9]

- **Performance:** Vite uses native ES Modules (ESM) during development, meaning the server starts instantly, regardless of app size.
- **HMR (Hot Module Replacement):** When you save a file, the change reflects instantly in the browser. In Atomic Design, where you are constantly tweaking small atoms, this speed is vital for developer experience.
- **Configuration:** Vite's configuration (vite.config.ts) allows us to set up "Path Aliases," which are essential for a deep atomic directory structure.

2.2 Directory Structure and Path Aliases

A strict directory structure is the physical manifestation of our Atomic philosophy. You will find the following structure in src/:

```
src/
├── assets/ # Static files (images, global fonts)
├── components/ # The Atomic Library
│ ├── atoms/ # Folder for each Atom (e.g., atoms/Button)
│ ├── molecules/ # Folder for each Molecule
│ ├── organisms/ # Folder for each Organism
│ └── templates/ # Layout definitions
├── pages/ # Route Controllers (Page Level)
├── hooks/ # Custom React Hooks (Logic atoms)
├── store/ # Atomic State (Jotai/Recoil atoms)
├── styles/ # Design Tokens and Global CSS
├── utils/ # Pure functions and helpers
└── main.tsx # Entry point
```

2.2.1 Configuring Aliases for Maintainability

In a professional project, we never use relative paths like import Button from '../../../../components/atoms/Button'. This is brittle. If you move a file, the path breaks. Instead, we configure **Path Aliases** in vite.config.ts.[2]

Current Project Configuration:

Alias	Path mapping	Usage Example
@atoms	./src/components/atoms	import { Button } from '@atoms/Button'
@molecules	./src/components/molecules	import { SearchBar } from '@molecules/SearchBar'
@organisms	./src/components/organisms	import { Header } from '@organisms/Header'
@templates	./src/components/templates	import { MainLayout } from '@templates/MainLayout'
@store	./src/store	import { userAtom } from '@store/user'
@hooks	./src/hooks	import { useClickOutside } from '@hooks/useClickOutside'

This configuration requires synchronization between vite.config.ts (for the bundler) and tsconfig.json (for the IDE/TypeScript). As an intern, you should always use these aliases. It makes the code readable and "move-safe."

2.3 Dependency Enforcement: Keeping the Hierarchy Clean

One of the most common mistakes new engineers make is "Hierarchy Violation." This occurs, for example, when an Atom imports an Organism.

- **Scenario:** You have a Button atom. You want the button to open the LoginModal (an organism). You import LoginModal into Button.
- **The Problem:** The Button is now coupled to the LoginModal. You can never use that button anywhere else without dragging the modal with it. It also creates a circular dependency if the Modal uses the Button.

To prevent this, we use **Dependency Cruiser** or ESLint rules (eslint-plugin-atomic-design).[11] These tools run during the build process and will block your code if you violate the hierarchy.

The Golden Rules of Dependency:

1. **Atoms** can only import -> External Libraries, Tokens, Utils.

2. **Molecules** can import -> Atoms, External Libraries, Tokens, Utils.
3. **Organisms** can import -> Molecules, Atoms, External Libraries, Tokens, Utils.
4. **Templates** can import -> Organisms, Molecules, Atoms.
5. **Pages** can import -> Templates, Organisms, Molecules, Atoms.

Part III: Detailed Component Architecture (Basic to Pro)

3.1 Atoms: The Art of the "Dumb" Component

An atom might seem simple, but writing a *good* atom is an art form. Atoms must be **Generic**, **Accessible**, and **Robust**.

3.1.1 The Anatomy of a Professional Atom

Let's look at the implementation of a Button atom.

TypeScript

```
// src/components/atoms/Button/Button.tsx
import React from 'react';
import { cva, type VariantProps } from 'class-variance-authority'; // Helper for class variants
import { Loader } from '@atoms/Loader'; // Sibling atom import
import styles from './Button.module.css';

// 1. Define Variants using a library or strict CSS classes
// This enforces consistency. You can't just pass "blue", you must pass "primary".
const buttonVariants = cva(styles.base, {
  variants: {
    intent: {
      primary: styles.primary,
      secondary: styles.secondary,
      danger: styles.danger,
    },
    size: {
      small: styles.small,
      medium: styles.medium,
      large: styles.large,
    },
  },
  defaultVariants: {
    intent: 'primary',
    size: 'medium',
  },
});

// 2. The Interface extends HTML Attributes
// PRO TIP: By extending React.ButtonHTMLAttributes, we automatically support
// onClick, onFocus, type, disabled, etc. without manually typing them.
interface ButtonProps
  extends React.ButtonHTMLAttributes<HTMLButtonElement>,
  VariantProps<typeof buttonVariants> {
  isLoading?: boolean;
  leftIcon?: React.ReactNode;
}

export const Button = React.forwardRef<HTMLButtonElement, ButtonProps>(
  ({ className, intent, size, isLoading, leftIcon, children, disabled,...props }, ref) => {
    return (
      <button
        ref={ref}
        className={buttonVariants({ intent, size, className })}
        disabled={disabled |

| isLoading} // PRO TIP: Handle loading state disabling
        {...props} // Spread remaining props (aria-labels, events)
      >
        {isLoading? (
          <Loader size="small" />
        ) : (
          <>
            {leftIcon && <span className={styles.icon}>{leftIcon}</span>}
            {children}
          </>
        )}
      </button>
    );
  }
);

Button.displayName = 'Button';
```

Insights for the Intern:

- **forwardRef:** Used so that parent components can access the DOM node if necessary (e.g., for focusing the button programmatically).
- **Props Spreading (...props):** This is essential for atoms. It allows a developer to use the button like a normal HTML button.
- **State:** Notice there is **no useState** here. The atom doesn't decide *when* to load; the parent tells it to load. This is a "Controlled Component" or "Dumb Component."

3.1.2 Design Tokens Integration

Atoms are the primary consumers of **Design Tokens**.[13] In our src/styles folder, we do not use hardcoded hex codes. We use token variables.

Why? If the brand rebrands from Blue to Purple, we change one token file, and every Atom updates automatically.

CSS

```css
/* src/styles/tokens.css */
:root {
  --color-primary: #0052cc;
  --color-danger: #ff0000;
  --spacing-sm: 8px;
  --spacing-md: 16px;
  --font-base: 'Inter', sans-serif;
}
```

In the component CSS:

CSS

```css
/* Button.module.css */
.primary {
  background-color: var(--color-primary);
  padding: var(--spacing-md);
}
```

3.2 Molecules: Managing Composition and Local State

Molecules are where we begin to see **Local State**. While Atoms are usually stateless, a Molecule often needs to manage UI-specific state that doesn't matter to the rest of the app.[7]

3.2.1 Case Study: The Search Bar

Consider a SearchBar molecule composed of an Input atom and a Button atom.

TypeScript

```
// src/components/molecules/SearchBar/SearchBar.tsx
import React, { useState } from 'react';
import { Input } from '@atoms/Input';
import { Button } from '@atoms/Button';
import { Icon } from '@atoms/Icon';
import styles from './SearchBar.module.css';

interface SearchBarProps {
  onSearch: (query: string) => void; // Callback prop
  placeholder?: string;
}

export const SearchBar: React.FC<SearchBarProps> = ({ onSearch, placeholder }) => {
  // Local State: Only this component cares about what is currently being typed.
  const = useState('');

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    onSearch(term); // Pass the data UP to the parent
  };

  return (
    <form className={styles.wrapper} onSubmit={handleSubmit}>
      <div className={styles.inputGroup}>
        <Icon name="search" className={styles.icon} />
        <Input
          value={term}
          onChange={(e) => setTerm(e.target.value)}
          placeholder={placeholder |
| "Search..."}
        />
      </div>
      <Button type="submit" intent="primary" disabled={!term}>
        Search
      </Button>
    </form>
  );
};
```

Insights for the Intern:

- **Responsibility:** The SearchBar handles the *act* of typing and submitting. It does **not** handle the logic of fetching results from an API. That is business logic. The molecule simply reports: "User wants to search for 'X'."
- **Encapsulation:** The layout (how the button sits next to the input) is defined here.

3.3 Organisms: Context and lists

Organisms often handle collections of data. A common organism is a List or Grid that renders many Molecules.

3.3.1 Case Study: The ProductGrid

TypeScript

```
// src/components/organisms/ProductGrid/ProductGrid.tsx
import React from 'react';
import { ProductCard } from '@molecules/ProductCard';
import { EmptyState } from '@atoms/EmptyState';
import { Grid } from '@atoms/Grid'; // A layout atom

interface Product {
  id: string;
  title: string;
  price: number;
  image: string;
}

interface ProductGridProps {
  products: Product;
  isLoading: boolean;
  onAddToCart: (id: string) => void;
}

export const ProductGrid: React.FC<ProductGridProps> = ({
  products,
  isLoading,
  onAddToCart
}) => {
  if (isLoading) return <Grid loading />; // Grid handles its own skeleton state

  if (products.length === 0) {
    return <EmptyState title="No products found" />;
  }

  return (
    <Grid columns={3}>
      {products.map((product) => (
        <ProductCard
          key={product.id}
          title={product.title}
          price={product.price}
          imageUrl={product.image}
          // The Organism mediates the event from the Molecule to the outside world
          onAction={() => onAddToCart(product.id)}
        />
      ))}
    </Grid>
  );
};
```

The "Pro" Nuance: Notice ProductGrid is still a **Presentational Component**. It receives products as a prop. It does not use fetch or useQuery internally. This makes it easy to test and reusable (e.g., "Similar Products" grid vs. "Main Search Results" grid).

Part IV: Data Flow and Atomic State Management

This is the most complex part of the onboarding. How does data move? We follow a **Unidirectional Data Flow**: Props go down, Events go up.

However, passing props down 15 levels (Prop Drilling) is bad. To solve this, we use **Atomic**

State Management. Since we are using an Atomic Design for UI, we use an Atomic Library for state: **Jotai** (or Recoil).[15]

4.1 Concept: Atoms of State

Just as we have Atoms of UI (Button), we have Atoms of State.

- **UI Atom:** A small, reusable visual element.
- **State Atom:** A small, reusable piece of data.

This is different from Redux, which has one giant "Store" object for the whole app. In Jotai, state is decentralized and composable.[16]

4.2 Implementing Jotai in Atomic Design

We keep our state definitions in the @store directory.

4.2.1 Creating State Atoms

TypeScript

```typescript
// src/store/cartAtoms.ts
import { atom } from 'jotai';

// 1. Primitive Atom: Holds the raw data
// This is like 'useState' but global
export const cartItemsAtom = atom<{ id: string; quantity: number }>();

// 2. Derived Atom (Read-Only): Computed data
// This is like a 'selector' or 'useMemo'
// It updates automatically when cartItemsAtom changes.
export const cartCountAtom = atom((get) => {
  const items = get(cartItemsAtom);
  return items.reduce((total, item) => total + item.quantity, 0);
});

// 3. Action Atom (Write-Only): Logic for modifying state
export const addToCartAtom = atom(
  null, // No read value
  (get, set, productId: string) => {
    const items = get(cartItemsAtom);
    const existing = items.find(i => i.id === productId);
```

```
   if (existing) {
     // Update logic
     const updated = items.map(i =>
       i.id === productId? {...i, quantity: i.quantity + 1 } : i
     );
     set(cartItemsAtom, updated);
   } else {
     set(cartItemsAtom, [...items, { id: productId, quantity: 1 }]);
   }
 }
);
```

4.3 Connecting State to Architecture

The critical rule: **Where do we use these atoms?**

- **Avoid:** Using useAtom inside **Atoms** or low-level **Molecules**. This couples them to the global state and makes them hard to reuse.
- **Prefer:** Using useAtom inside **Organisms** (sometimes) and **Pages** (always).

4.3.1 The "Container/View" Pattern in Atomic Design

We often separate the logic (Container) from the look (View).

The Page (Container/Controller):

TypeScript

```
// src/pages/ProductPage.tsx
import { useAtom } from 'jotai';
import { productsQueryAtom } from '@store/products'; // Async state
import { addToCartAtom } from '@store/cartAtoms';
import { ProductGrid } from '@organisms/ProductGrid';
import { MainLayout } from '@templates/MainLayout';

export const ProductPage = () => {
 // The Page handles the "Smart" logic
 const [products] = useAtom(productsQueryAtom);
 const = useAtom(addToCartAtom);
 return (
  <MainLayout title="Shop">
   <ProductGrid
    products={products}
    onAddToCart={addToCart}
    isLoading={!products}
   />
  </MainLayout>
 );
};
```

By doing this, the ProductGrid Organism remains pure. It receives data via props. The ProductPage acts as the conductor, fetching data from the Jotai atoms and passing it down.

4.4 Data Flow Summary Table

Data Type	Direction	Mechanism	Atomic Level
UI State (e.g., input value, menu open)	Local	useState	Molecules / Organisms
Business Data (e.g., User, Products)	Top-Down	Props (from Page)	Organisms -> Molecules
Events (e.g., Click, Submit)	Bottom-Up	Callbacks / Handlers	Atoms -> Molecules -> Organisms
Global State (e.g., Cart, Theme)	Sidebar/Global	Jotai/Recoil Atoms	Pages / High-level Organisms

---

Part V: Professional Workflows & Best Practices

To work at a "Pro" level, writing code is not enough. You must document, test, and maintain it.

5.1 Storybook: Visual Documentation

Since we build components in isolation, we need a way to view them in isolation. **Storybook** is our workshop.[17] For every component you build, you must write a "Story."

Why?

1. **Catalog:** It acts as a visual inventory. You can see all available buttons without searching code.
2. **Edge Cases:** You can create stories for "Long Text," "Error State," "Loading State" to verify the component breaks gracefully.

Example Story:

TypeScript

```typescript
// src/components/atoms/Button/Button.stories.tsx
import type { Meta, StoryObj } from '@storybook/react';
import { Button } from './Button';

const meta: Meta<typeof Button> = {
  title: 'Atoms/Button', // Creates the folder structure in Storybook sidebar
  component: Button,
  tags: ['autodocs'], // Generates documentation page automatically
  argTypes: {
    intent: { control: 'select', options: ['primary', 'secondary'] },
  },
};

export default meta;
type Story = StoryObj<typeof Button>;

export const Primary: Story = {
  args: {
    intent: 'primary',
    children: 'Confirm Action',
  },
};

export const Loading: Story = {
  args: {
    intent: 'primary',
    isLoading: true,
    children: 'Confirm Action',
  },
};
```

5.2 Testing Strategy: The Pyramid

In Atomic Design, testing is granular.[19]

Component Level	Test Type	Tool	Focus
Atoms	Unit Test	Vitest / RTL	Rendering, Props, Event firing. (Does clicking call onClick?)
Molecules	Unit/Integration	Vitest / RTL	Interaction. (Does typing in input update value? Does submit button trigger form?)
Organisms	Integration	Vitest / RTL	Data display. (Does the list map the items correctly? Does empty state show?)
Pages	E2E (End-to-End)	Playwright / Cypress	User Flows. (Can I login and see the dashboard?)

Pro Tip for Interns: Don't test that React works. Don't test useState. Test *user behavior*.

- *Bad:* expect(state.value).toBe('hello')
- *Good:* fireEvent.change(input, {target: {value: 'hello'}}); expect(input).toHaveValue('hello');

5.3 Handling Cross-Cutting Concerns

Some things don't fit neatly into the hierarchy, such as Authentication, Logging, and Theming. These are "Cross-Cutting Concerns".[21]

- **Theming:** Handled via CSS Variables (Tokens) injected at the root App.tsx level.
- **Authentication:** Managed via a high-level AuthProvider (Context) or a User Atom in Jotai. It wraps the entire application in main.tsx.

- **Utils:** Pure functions (e.g., formatDate, currencyConverter) live in src/utils. They can be imported by any layer.

5.4 Refactoring: The Component Lifecycle

A component is not static. It evolves.

1. **Birth:** You create a Card in molecules.
2. **Growth:** You add more features. A "Buy" button. A "Like" counter. It's getting complex.
3. **Promotion:** The Card now contains other molecules. It should be moved to organisms.
4. **Extraction:** You notice the "Buy Button" logic is complex. You extract it into a separate molecule.

When to promote a Molecule to an Organism?

- When it starts managing business state (e.g., fetching data).
- When it contains other molecules.
- When it becomes specific to a section of the site (e.g., HomePageHero vs just Hero).

Part VI: Conclusion and Onboarding Checklist

Mastering Atomic Architecture is about discipline. It prevents the "spaghetti code" that plagues large React applications. By respecting the boundaries of Atoms, Molecules, and Organisms, and by utilizing Atomic State (Jotai) to manage data flow, we create a codebase that is predictable and scalable.

6.1 Your First Week Checklist

To verify your understanding, complete the following tasks:

1. [ ] **Explore Storybook:** Browse the existing Atoms and Molecules. Familiarize yourself with the available props.
2. [ ] **Read the Tokens:** Look at src/styles/tokens.ts. Memorize the spacing units (e.g., sm, md, lg) so you don't hardcode pixels.
3. [ ] **Build an Atom:** Create a <Badge> atom (displays a small status label). Add variants for "Success", "Warning", "Error". Write a Story for it.
4. [ ] **Build a Molecule:** Create a UserBadge molecule using your <Badge> and the existing <Avatar> atom.
5. [ ] **Review Dependencies:** Check vite.config.ts to understand the aliases.

Welcome to the team. You are now ready to build systems, not just pages.

End of Manual

Works cited

1. Atomic Design Methodology, accessed December 22, 2025, https://atomicdesign.bradfrost.com/chapter-2/
2. Mastering Modular Architecture with React and Atomic Design: Advanced Techniques and Hands-On Examples | by Abdulnasır Olcan | Medium, accessed December 22, 2025, https://medium.com/@abdulnasirolcan/mastering-modular-architecture-with-react-and-atomic-design-advanced-techniques-and-hands-on-93e649654a06
3. Atomic Design Pattern: How to set up your Reactjs Project Structure? - DEV Community, accessed December 22, 2025, https://dev.to/mroman7/atomic-design-pattern-how-to-set-up-your-reactjs-project-structure-44pm
4. danilowoz/react-atomic-design: How the Atomic Design ... - GitHub, accessed December 22, 2025, https://github.com/danilowoz/react-atomic-design
5. Atomic Architecture in React for Designing Components - Angular Minds, accessed December 22, 2025, https://www.angularminds.com/blog/atomic-architecture-react-for-component-design
6. Atomic Design in React: Best Practices - Propelius Technologies, accessed December 22, 2025, https://propelius.ai/blogs/atomic-design-in-react-best-practices
7. Thinking About React, Atomically | by Katia Wheeler - Medium, accessed December 22, 2025, https://medium.com/@wheeler.katia/thinking-about-react-atomically-608c865d2262
8. Atomic Design in React: Building Scalable Component Systems | by Rotimi balogun, accessed December 22, 2025, https://medium.com/@shefiub0/atomic-design-in-react-building-scalable-component-systems-76d3f977e10c
9. What is your preferred folder structure for CRA React Typescript app? Best Practices? : r/reactjs - Reddit, accessed December 22, 2025, https://www.reddit.com/r/reactjs/comments/151q4d2/what_is_your_preferred_folder_structure_for_cra/
10. Setting |Up Path Aliases in Vite for a Cleaner Project Structure | by Khusan Mirobidov, accessed December 22, 2025, https://medium.com/@husanmirobidov/setting-up-path-aliases-in-vite-for-a-cleaner-project-structure-93002608d8b4
11. prowner/eslint-plugin-atomic-design-hierarchy - GitHub, accessed December 22, 2025, https://github.com/prowner/eslint-plugin-atomic-design-hierarchy
12. Taking Frontend Architecture Serious With Dependency-cruiser - Xebia, accessed December 22, 2025, https://xebia.com/blog/taking-frontend-architecture-serious-with-dependency-cruiser/
13. accessed December 22, 2025,

https://www.designsystemscollective.com/a-comprehensive-guide-to-atomic-design-and-design-tokens-in-modern-ui-ux-development-288a996a483a#:~:text=When%20Atomic%20Design%20and%20Design,components%20maintain%20consistency%20and%20flexibility.

14. Design Systems in React – Atomic Design (Part 1) - Buszewski.com, accessed December 22, 2025, https://www.buszewski.com/writings/2024-09-23-design-systems-in-react-atomic-design-part-1/
15. Comparison — Jotai, primitive and flexible state management for React, accessed December 22, 2025, https://jotai.org/docs/basics/comparison
16. Using Jotai in Your React Application - Bits and Pieces, accessed December 22, 2025, https://blog.bitsrc.io/using-jotai-in-your-react-application-de460568ac9d
17. Storybook for React with Vite, accessed December 22, 2025, https://storybook.js.org/docs/get-started/frameworks/react-vite
18. Atomic Design in React | Storybook in React - YouTube, accessed December 22, 2025, https://www.youtube.com/watch?v=zaaknDxK-Fg
19. Component Testing | Guide - Vitest, accessed December 22, 2025, https://vitest.dev/guide/browser/component-testing
20. Mastering Unit Testing in React: Best Practices for Efficient Tests with Vitest and React testing library | by Victor Rillo | Medium, accessed December 22, 2025, https://medium.com/@victorrillo/mastering-unit-testing-in-react-best-practices-for-efficient-tests-with-vitest-and-react-testing-181408af7a10
21. design patterns - Cross cutting concern example - Stack Overflow, accessed December 22, 2025, https://stackoverflow.com/questions/23700540/cross-cutting-concern-example