The Three Pillars of React Native UI

Mastering Structure, Style, and Interaction to Build Polished Mobile Applications

Every Great UI Rests on These Three Pillars

Structure (The 'What')

The architectural foundation. How components are organized, from their core logic to the fundamental building blocks like `View` and `Text`.

Style (The 'Look')

The visual language. How to to apply a consistent and maintainable look and feel, and the trade-offs between different styling methods.

Interaction (The 'How')

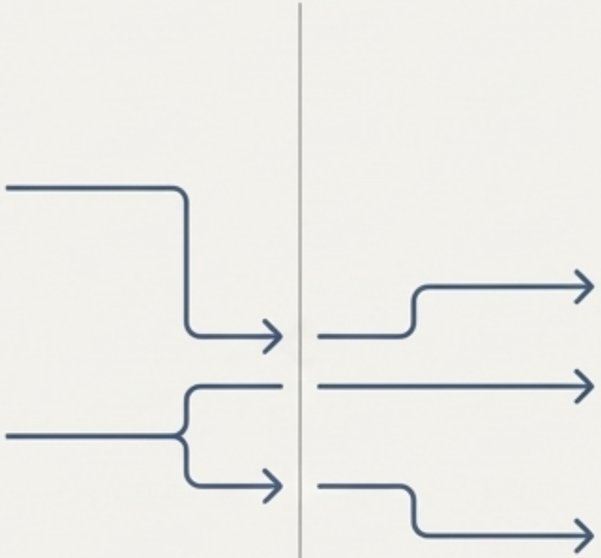
The user connection. How the application responds to touch, gestures, and input to create a living, breathing experience.

The Philosophy: Separating the Brains from the Beauty

In React Native, we separate components into two roles to create scalable, reusable, and maintainable code.

Smart Component (Container)

Manages the logic, state, and data flow. It knows *how things work*.



Dumb Component (Presentational)

Renders the UI. It only knows *what things look like* based on the data (props) it's given.

Pillar I: Structure

Smart vs. Dumb: A Practical Comparison

Smart Component	Dumb Component
Primary Responsibility: Logic and data flow.	**Primary Responsibility:** Presenting the UI.
State Management: Yes. Manages its own state.	**State Management:** No. Receives all data via props.
Data Source: Fetches data, connects to state management.	**Data Source:** Receives data from parent components via `props`.
UI Rendering: Indirectly, by passing data down to Dumb components.	**UI Rendering:** Directly. Its main purpose.
Reusability: Lower, often tied to a specific application context.	**Reusability:** High. Can be used anywhere as long as the correct props are provided.

Key Takeaway: This separation makes code easier to test, reuse, and reason about.

Pillar I: Structure

The Fundamental Building Blocks: View and Text

Every React Native UI is built from `View` and `Text`. These components map directly to the platform's native UI elements, ensuring performance and a native feel.

<View>

Analogy

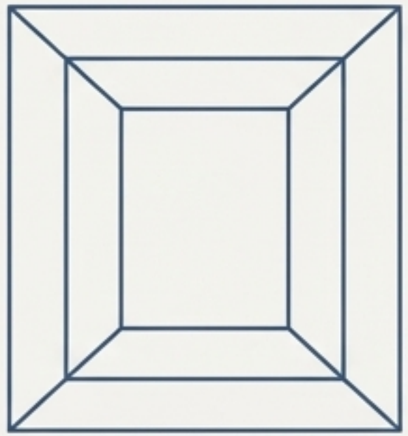
The ` of the mobile world.

Purpose

The universal container for layout, styling, and grouping other elements.

Key Features

Supports Flexbox for responsive layouts (`justifyContent`, `alignItems`), can nest other elements, and handles touch interactions.



<Text>

Analogy

The <p> or ` of mobile.

Purpose

The *only* component for displaying strings of text.

Key Features

Manages text wrapping and styling. Nested <Text> elements can inherit styles from their parent.

Critical Rule: All text, no matter how short, *must* be wrapped in a <Text> component. You cannot place raw text inside a <View>.

Pillar II: Style

Defining the Look: StyleSheet API vs. Inline Styles

In React Native, styling is done with JavaScript objects that mimic CSS properties. The key decision is ***where*** you define these objects.

The Organized Approach

```
const Component = () => (
  <View style={styles.container}>
    <Text style={styles.title}>Hello World</Text>
  </View>
);

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
  },
});
```

The On-the-Fly Approach

```
const Component = () => (
  <View style={{ flex: 1, backgroundColor: '#fff' }}>
    <Text style={{ fontSize: 24, fontWeight: 'bold' }}>
      Hello World
    </Text>
  </View>
);
```

Key Insight: While both methods can produce the same visual result, they have significant implications for the health of your codebase.

Pillar II: Style

The Stylist’s Dilemma: Choosing the Right Tool for the Job

Aspect	StyleSheet API	Inline Style
Definition	Defined once outside JSX with `StyleSheet.create()`	Defined directly in the JSX `style` prop
Readability	**High**: Separates style from component logic	**Low**: Mixes presentation logic directly into JSX
Reusability	**High**: Can be exported and shared across components	**Low**: Tied to a single element instance
Performance	**Optimized**: Sent to the native side only once	Re-evaluated on every render
Best For	✅ Production apps, design systems, reusable components	⚠️ Quick prototyping, dynamic styles based on state/props

Golden Rule: **Default to** `StyleSheet.create()`. Use inline styles only for dynamic values that must be computed during render.

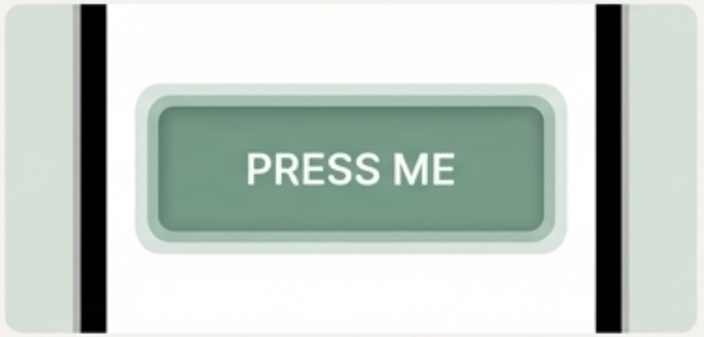
Pillar III: Interaction

Bringing the UI to Life with User Input

A great UI must provide clear and immediate feedback. React Native provides a suite of components to detect touches and capture user input, forming the bridge between the user and your app's logic.

TouchableOpacity

Provides feedback by fading its child's opacity. The lightweight workhorse for most buttons and interactive elements.



TouchableHighlight

Provides feedback by changing the background color. Ideal for lists and elements that need a more pronounced selection state.

TextInput

The essential component for capturing text from the user via the native keyboard.

Pillar III: Interaction

Touch Feedback: Fading Opacity vs. Highlighting Background

TouchableOpacity

- **Visual Feedback:** Reduces the opacity of its child element on press.
- **Use Cases:** Standard buttons, icons, image-based links. Any time a subtle, non-intrusive feedback is needed.
- **Performance:** Highly performant and versatile.

TouchableHighlight

- **Visual Feedback:** Darkens the background by applying an `underlayColor` on press.
- **Use Cases:** Rows in a list, navigation items, buttons within a toolbar where a clear selection state is critical.
- **Customization:** Offers more control over the pressed state's appearance.

Pro-Tip: Always provide visible feedback for any touchable element. It's fundamental to a good user experience.

Pillar III: Interaction

Capturing Data with Controlled TextInputs

`TextInput` is the gateway for user-provided text. To build reliable forms, you must 'control' the component by connecting it to your component's state.

Key Props for State Management

- `value`: Binds the input's displayed text directly to a state variable.
- `onChangeText`: A callback function that updates the state variable every time the user types.

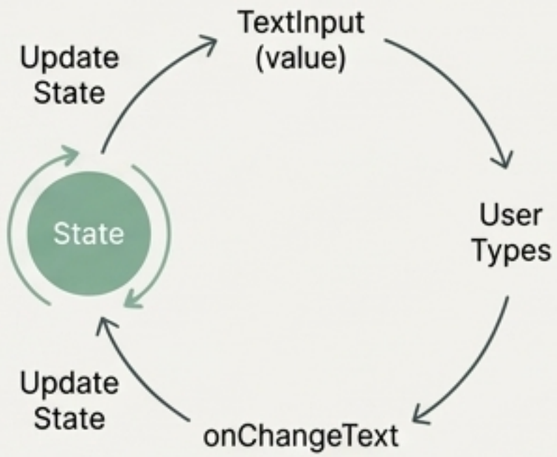
```
// In your component
const [name, setName] = useState(''); // 1. Initialize state

// In your JSX
<TextInput
  placeholder="Enter your name"
  value={name} // 2. Bind input value to state
  onChangeText={newName => setName(newName)} // 3. Update state on change
/>
```

A piece of state is created to hold the input's value. It starts as an empty string.

The `TextInput`'s display is locked to the current value of the `name` state variable.

When the user types, this function is called. It updates the state with the new text, which causes the component to re-render and display the new `value`.



Synthesis I: Anatomy of a 'Dumb' Component

Case Study: The Book Card

The Code (BookCard.js)

```
const BookCard = ({ title, author, year }) => (
  <View style={styles.container}>
    <Text style={styles.title}>{title}</Text>
    <Text style={styles.author}>{author}</Text>
    <Text style={styles.year}>{year}</Text>
  </View>
);

const styles = StyleSheet.create({
  // ... styles for container, title, etc.
});
```

Data Input. This component is “Dumb”—it only displays the data it receives.

Style. Using the `StyleSheet` API for clean, reusable, and performant styling.

Structure. Using fundamental blocks to create a clear layout.

The Result

The Hitchhiker's Guide to the Galaxy

Douglas Adams

1979

Key Insight: This component perfectly executes its single responsibility: presenting data. It has no internal logic or state, making it highly reusable and easy to test.

Synthesis II: From Blueprint to Reality

Case Study: The BMI Calculator

BMICalculator.js

```
import React, { useState } from 'react';
import { StyleSheet, View, Text, TextInput, TouchableOpacity } from 'react-native';

const BRICalculator = () => {
  const [weight, setReight] = useState('');
  const [height, setHeight] = useState('');
  const [bmi, setBmi] = useState(null);

  const calculateBNI = () => {
    // Calculatien logic
    const heightMeters = height / 100;
    const calculatedBwi = weight / (heightNeters * heightNeters);
    setBwi(calculatedBwi.toFlxed(1));
  };

  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder="Weight (kg)"
        value={weight}
        onChangeText={setNeight}
        keyboardType="numer1c"
      />
      <TextInput
        style={styles.input}
        placeholder="He1ght (cm)"
        value={height}
        onChangeText={setHeight}
        keyboardType="numer1c"
      />
      <TouchableOpacity style={styles.button} onPress={calculateBNI}>
        <Text style={styles.buttonText}>Calculatex/Text>
      </TouchableOpacity>

      {bmi && (
        <Text style={styles.result}>Your BNI is: {bmi}</Text>
      )}
    </View>
  );
};

export default BMICalculator;
```

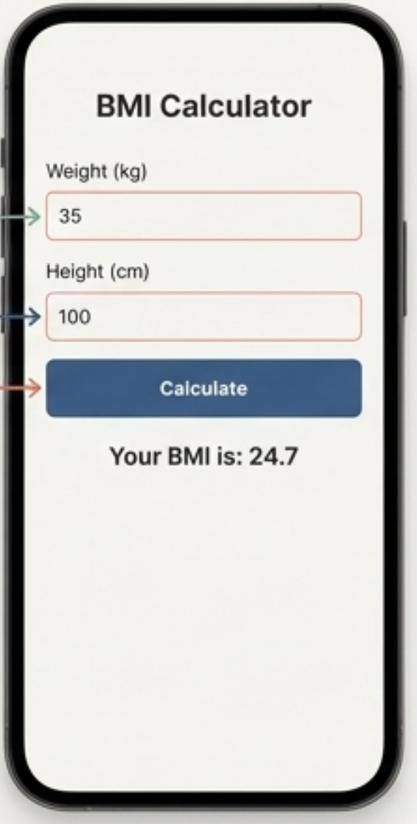
Interaction* Managing user input and calculated results in state.

Interaction*. Controlled components for capturing data.

Structure*. Organizing the form and result display.

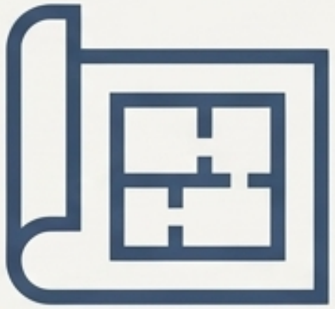
Interaction*. Triggering the calculation logic.

Style*. Centralizing all styling rules for a clean and maintainable component.



$$BMI = \frac{weight\ (kg)}{(height\ (m))^2}$$

The Core Principles of React Native UI



Structure

Separate logic (Smart) from presentation (Dumb). Build your foundation with `View` and `Text`.

Style

Prioritize `StyleSheet` for maintainable, performant, and clean styling. Use inline styles with intention, not as a default.

Interaction

Provide clear, responsive feedback for every user action. Control your inputs with state to create predictable and reliable forms.

Knowledge Reinforced

Q1: What is the primary reason to separate Smart and Dumb components?

A: To achieve “**separation of concerns.**” It makes your code more testable, reusable, and easier for teams to maintain by isolating logic from presentation.

Q2: When is an inline style the right choice over StyleSheet?

A: Only when a style property depends on a dynamic value calculated during render (e.g., from state or props). For all static styles, StyleSheet is preferred.

Q3: What's the key difference in user feedback between TouchableOpacity and TouchableHighlight?

A: TouchableOpacity provides subtle feedback by fading opacity. TouchableHighlight provides stronger feedback by changing the background color, which is ideal for highlighting selections.