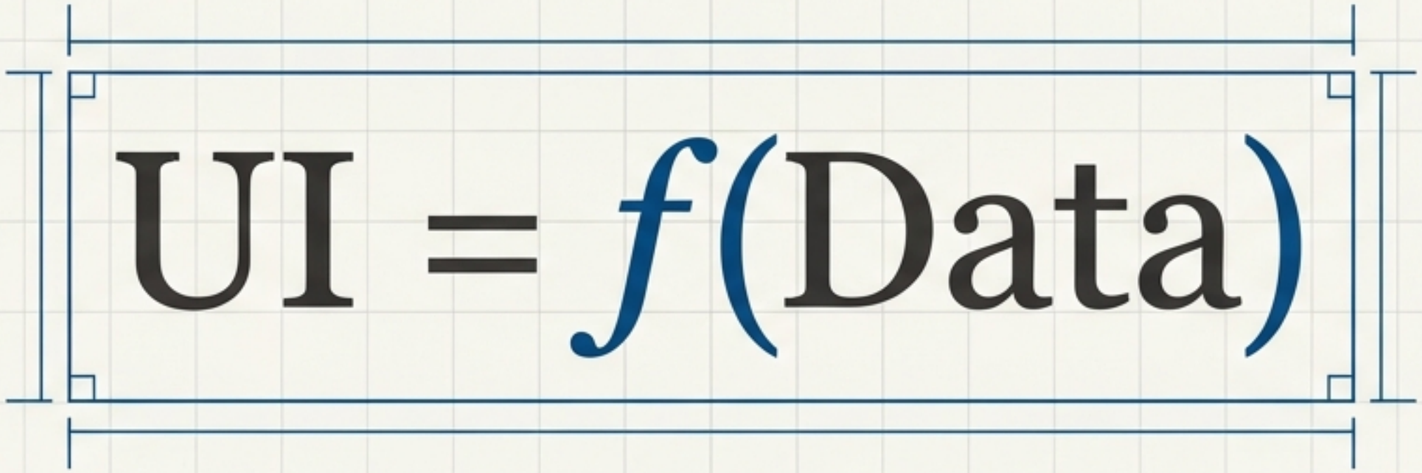
# Introduction to React Native Programming

## Architecting Modern Mobile Applications




React Native represents a paradigm shift from imperative to declarative programming. Instead of describing *how* a system works, developers define *what* the UI should look like for a given set of data. This presentation deconstructs the architectural principles that make this possible.

# The Theoretical Foundations of React Native

React Native is not a single invention, but a convergence of multiple disciplines of software design. These principles work in harmony to create modular, maintainable, and predictable systems.

## Software Engineering

Principles of creating modular and maintainable systems.

## Object-Oriented Design

The power of encapsulation and clear separation of responsibilities.

## The Functional Paradigm

The mathematical purity of treating UI as a pure function of its data.

# Assembling the Development Toolkit

Before we build, we must gather our tools. The following are essential for a modern React Native workflow using Expo.

**Node.js (LTS):** Install the Long-Term Support version (e.g., 18 or 20) from nodejs.org. Verify with `node -v`.

**Local Expo CLI:** The global `expo-cli` is deprecated. Use the local version for every project via `npx expo`. To create a new project: `npx create-expo-app my-app`.

**Expo Go App:** Install from the App Store or Google Play. This is your live preview device.

**Code Editor:** Your IDE of choice (e.g., VS Code).


**(Optional) Android Studio:** Required for running on a local Android Emulator. Involves setting up the SDK and AVD managers.

**Key Command to remember:** To validate your complete environment, run `npx expo-doctor`.

# The Component: The Building Block of UI

## Conceptual Explanation

### Definition

A Component is a reusable, self-contained unit that defines both the structure (the what) and the logic (the how) of a UI segment.

### Analogy

Think of it as a 'conceptual module' — a small, independent part of a larger digital system.

$$UI = f(Data)$$

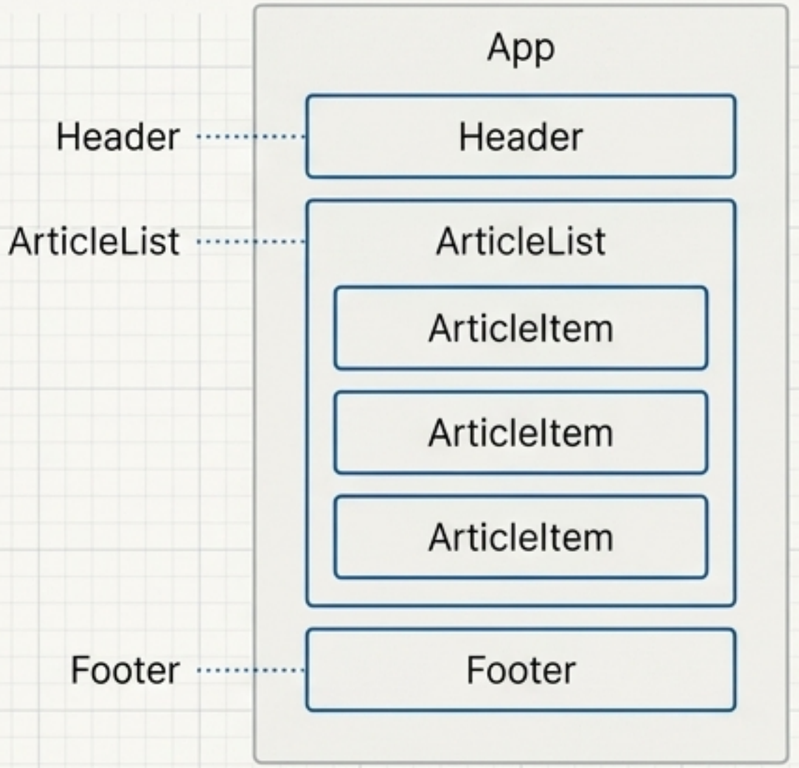
## Code Example

```
import { Text, View } from 'react-native';

const WelcomeMessage = () => {
  return (
    <View>
      <Text>Hello, World!</Text>
    </View>
  );
};

export default WelcomeMessage;
```

This is a **Stateless Functional Component**. It demonstrates *functional purity*—it will always return the same output for the same input, ensuring predictable behavior.

# Composition: Building Hierarchies from Simple Blocks

Composition allows smaller, simpler components to be combined to form larger, more complex UI structures. This is a central tenet of **Component-based Architecture** and promotes **Separation of Concerns**.

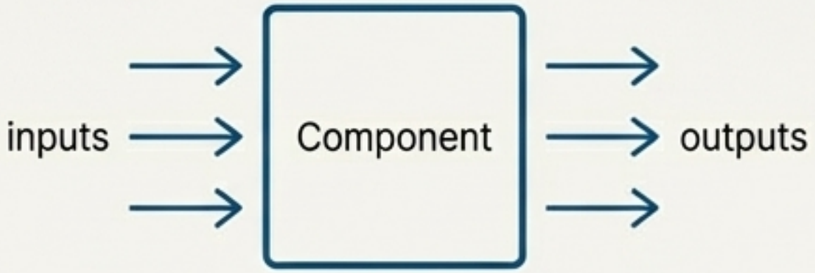



```
import WelcomeMessage from './WelcomeMessage';

const App = () => {
  return (
    <View>
      <Header />
      <WelcomeMessage />
      <Footer />
    </View>
  );
};
```

The `App` component composes the `Header`, `WelcomeMessage`, and `Footer` components into a complete screen.

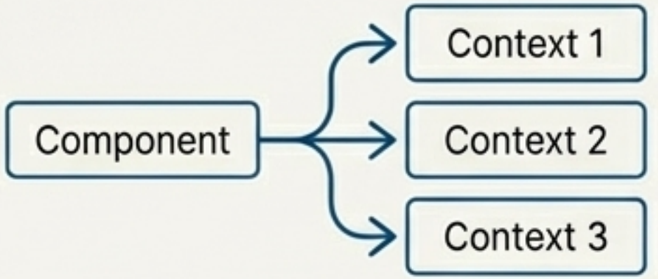
# The Four Key Characteristics of a Well-Designed Component

## Encapsulation



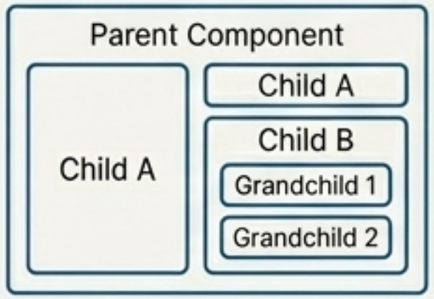

Components are self-contained. Their internal logic and data are hidden from the outside world. This makes them predictable and safe to use.

## Reusability



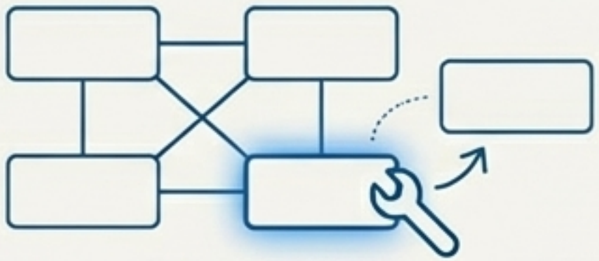

Components are designed to be used in multiple places with different data. This encourages **Parametric Polymorphism**—their behavior changes based on their inputs.

## Composition




Components can contain other components, allowing for the creation of complex UI hierarchies from simple, manageable parts.

## Maintainability



Because they are encapsulated, components can be updated, fixed, or replaced independently without causing unintended side effects elsewhere in the application.

# JSX: The Declarative Language for UI Architecture

## Concept

### What is JSX (JavaScript XML)?

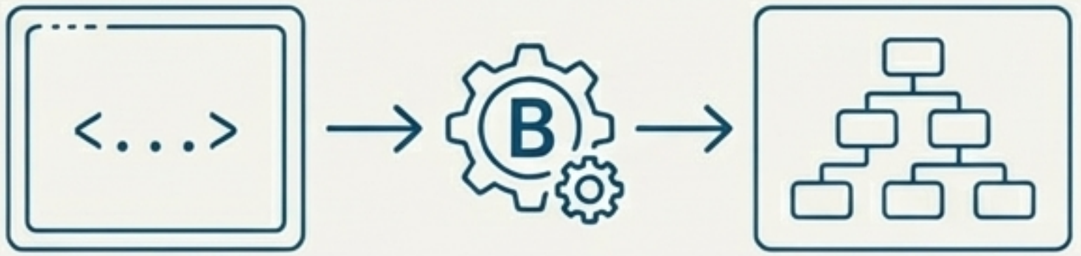
JSX is a syntax extension for JavaScript that allows you to write UI structure directly inside your code. It blends logic and presentation seamlessly.

**The Philosophy:** It embodies **Declarative Programming**. You tell the system *what to show*, not *how to draw it*.

## Structure and Theory

```
const name = "Student";
const element = <Text>Hello, {name}</Text>;
```

- "<View> and <Text> are React Native components that map to native UI elements (like `UIView` or `ViewGroup`).
- The {} curly braces allow you to embed any JavaScript expression directly within the markup.



# Declarative in Practice: The UI Reacts to Data

The true power of the declarative model is that the UI automatically updates when the underlying data changes. You don't manage the drawing; you only manage the data.

### Imperative (The 'Old Way')

```
// Psuedo-code
if (isLoggedIn) {
  show(welcomeMessage);
  hide(loginButton);
} else {
  hide(welcomeMessage);
  show(loginButton);
}
```

Manually describing each step to manipulate the UI.

### Declarative (The React Way)

```
function AuthUI({ isLoggedIn }) {
  return (
    <View>
      {isLoggedIn ? <WelcomeMessage /> :
       <LoginButton />}
    </View>
  );
}
```

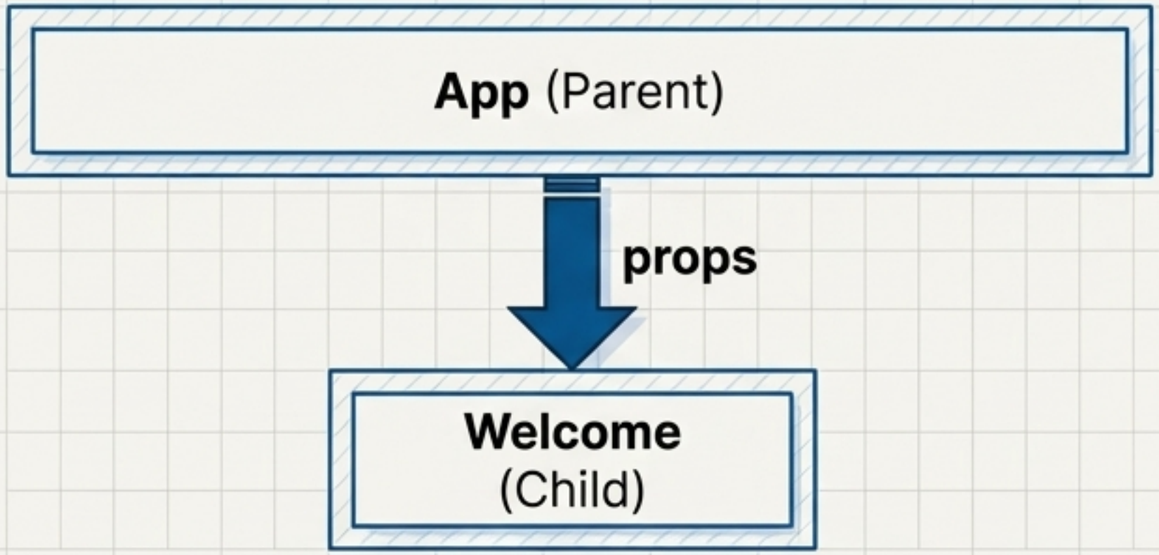
Describing what the UI should look like for a given state. React handles the rest.

# Data Mechanism I: Props for Configuration

## Core Concept

**Props** (short for properties) are **inputs** passed from a parent component to a child component. They are used to configure the child's appearance and behavior.

**Key Principle:** Props establish **unidirectional data flow**. Data flows from the top (parent) down (child) only.




```
// Parent Component
const App = () => {
  return <Welcome name="Alice" />;
};

// Child Component
const Welcome = (props) => {
  return <Text>Hello, {props.name}</Text>;
};
```

## Theoretical Connection

This reflects **Functional Programming**. Given the same `props`, the component will always produce the same output. It also promotes **Immutability**, as a child component cannot modify the props it receives.

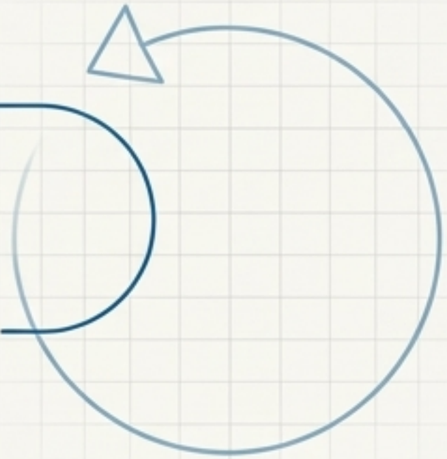
# Data Mechanism II: State for Internal Dynamics

## Core Concept

**State** represents the internal, changeable data that belongs to a single component. It is managed entirely within that component.

> **Key Principle:** When a component's state changes, React automatically **re-renders** the component and its children to reflect the new data.

```
import { useState } from 'react';
import { Text, Button } from 'react-native';

const Counter = () => {
  const [count, setCount] = useState(0);

  return (
    <View>
      <Text>You clicked {count} times</Text>
      <Button
         onPress={() => setCount(count + 1)}
         title="Click me"
      />
    </View>
  );
};
```



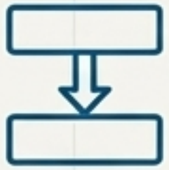
### The Philosophy

This is the essence of **Reactive Programming**. The state *drives* the UI. Data directly controls the interface.

- The `count` variable is a piece of state. Calling `setCount` updates it and triggers a UI re-render.

# Props vs. State: A Quick Reference

## PROPS



**Source**
External. Passed down from a parent component.

**Changeability**
Immutable. The component cannot change its own props.

**Purpose**
To configure a component from the outside.

**UI Update**
A change in props (from the parent) will trigger a re-render.

## STATE

**Source**
Internal. Managed and owned by the component itself.

**Changeability**
Mutable. The component can change its own state via `setState`.

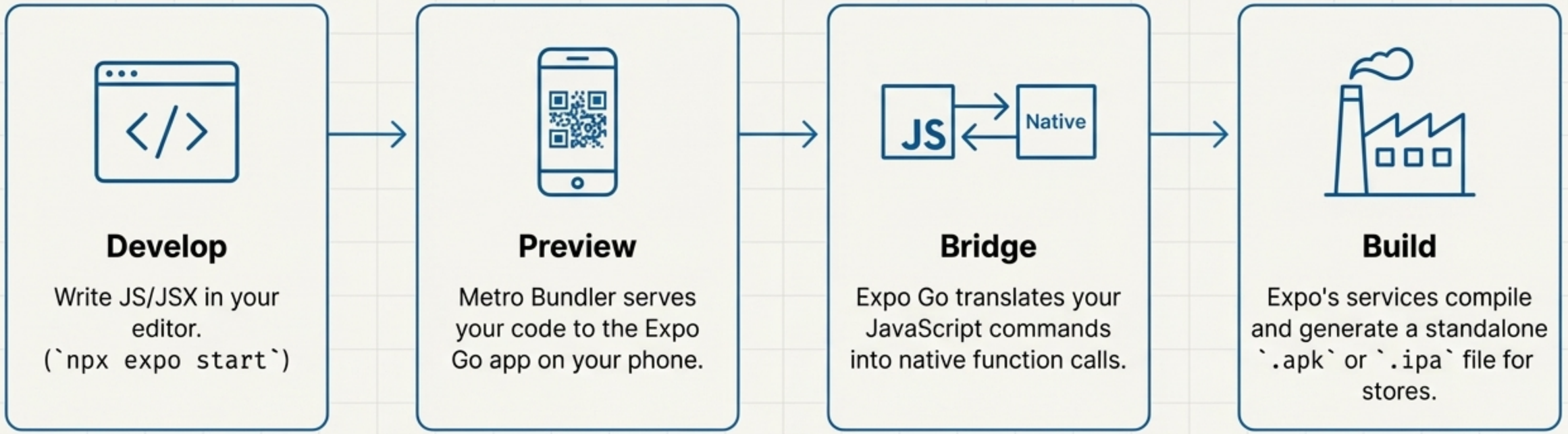
**Purpose**
To manage dynamic data that changes over time.

**UI Update**
A change in state (from within) will trigger a re-render.

# Expo: The Abstraction Layer Between Code and Native

**Core Concept: Expo** is a runtime environment and toolchain that simplifies building, running, and testing React Native apps. It removes the need for complex, manual configuration of Android and iOS SDKs.

**Theoretical Role:** Expo acts as an **Abstraction Layer**, serving as a bridge between your JavaScript logic and the underlying native operating systems.



# The Anatomy of an Expo Project

A new Expo project provides a clean, logical structure that reflects principles of **Modular Design** and **Scalability.**


The main entry point of your application. This is the root component of your UI tree.

Project metadata. It lists all dependencies (like React, React Native, and other libraries).

A directory for static assets like images, fonts, and sound files.

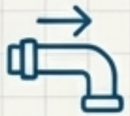
A directory you create to store your reusable UI components, promoting a modular architecture.

Contains all the downloaded dependency code. Managed by npm/yarn.

Project metadata. It lists all dependencies (like React,

# Synthesizing Theory and Practice

React Native embodies the harmony between abstract software theory and practical, real-world implementation.

| Theoretical Concept | Manifestation in Code & Tools |
| --- | --- |
|  Component-based Architecture | UI is divided into encapsulated, reusable units (`<MyComponent />`). |
|  Declarative Programming | JSX is used to describe the desired UI state (`<Text>Hello</Text>`). |
|  State-driven Rendering | The UI automatically updates when `useState` is called. |
|  Unidirectional Data Flow | Data is passed down from parent to child via `props`. |
|  Abstraction Layer | Expo bridges your JavaScript with the native OS. |

# From Theory to Interactive Reality

- React Native is theory *made tangible*.
- It merges the disciplines of **Software Architecture** and **Reactive Design**.
- Core principles like **Abstraction**, **Encapsulation**, and **Data Flow** are not just academic—they are what create robust, maintainable software systems.

**“React Native transforms software theory into interactive reality.”**