Modulo 2 — JSX e Componenti React Native

JSX • Componenti • Props e State • Styling • Liste • Hook • Mini To-Do

Obiettivi del modulo

Cosa saprai fare alla fine

- Scrivere JSX corretto (tag, props, espressioni).
- Creare componenti riutilizzabili e comporli.
- Passare dati con props e gestire lo stato con hook.
- Applicare stile con StyleSheet e Flexbox.
- Renderizzare liste con ScrollView e FlatList.
- Costruire una mini To-Do app (add/toggle/remove).

JSX

Che cos’è e perché si usa

- JSX è una sintassi che ti fa descrivere la UI con tag dentro JS/TS.
- In React Native non è HTML: usi componenti nativi (View, Text, Image…).
- JSX viene trasformato (transpile) in chiamate JavaScript per React.
- Vantaggio: UI leggibile e facile da comporre in componenti.

JSX: esempio minimo

Struttura base di un componente

Spiegazione

- Un componente è una funzione che ritorna JSX.
- In React Native non usi div/span, ma View/Text.
- Il return deve essere un albero con una radice.
- Il testo visibile deve stare dentro <Text>.

App.tsx

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

export default function App() {
  return (
    <View>
      <Text>Ciao React
Native!</Text>
    </View>
  );
}
```

JSX: regole pratiche

Errori comuni

- Testo sempre dentro <Text> (non direttamente in <View>).
- Props come attributi: <MyComp title="Ciao" />.
- Per inserire JS usa { ... } (graffe).
- Tag auto-chiusi quando serve: <Image />.
- Se un JSX diventa lungo, estrai in componenti.

JSX: espressioni con { }

Variabili e calcoli

Spiegazione

- Dentro { } puoi usare variabili, funzioni e calcoli.
- Non puoi mettere ‘if’ direttamente: usa ternari o &&.
- Calcoli complessi meglio farli sopra al return.

Esempio

```
const name = "Matteo";
const items = 3;

return (
  <View>
    <Text>Ciao {name}!</Text>
    <Text>Hai {items * 2}
punti.</Text>
  </View>
);
```

Rendering condizionale

Ternario e &&

Spiegazione

- Ternario: condizione ? A : B.
- AND: condizione && A (mostra A solo se true).
- Quando le condizioni aumentano, estrai in funzioni.

Condizioni

```
return (
  <View>
    {isLoggedIn ? (
      <Text>Benvenuto!</Text>
    ) : (
      <Text>Accedi per
continuare</Text>
    )}

    {hasError && <Text>Si è
verificato un errore</Text>}
  </View>
);
```

Componenti

Mattoni riutilizzabili

- Un componente incapsula UI, comportamento e stile.
- Composizione: componenti piccoli costruiscono componenti più grandi.
- Riutilizzo: stesso componente con props diverse.
- Componenti piccoli = manutenzione più semplice.

Componenti: Card riutilizzabile

Props tipizzate in TypeScript

Spiegazione

- Props rendono il componente configurabile.
- StyleSheet mantiene stili ordinati.
- Pattern utile per schermate reali.

components/InfoCard.tsx

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

type InfoCardProps = {
  title: string;
  description: string;
};

export function InfoCard({ title,
description }: InfoCardProps) {
  return (
    <View style={styles.card}>
      <Text style={styles.title}>{title}</Text>
      <Text style={styles.desc}>{description}
</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  card: { padding: 12, borderRadius: 12,
backgroundColor: "#fff" },
  title: { fontSize: 18, fontWeight: "600" },
  desc: { marginTop: 6, opacity: 0.8 },
});
```

Funzionali vs classe

Differenze (concetto)

- Funzionali: funzioni + hook (useState, useEffect…).
- Classi: class + this.state + lifecycle methods.
- Oggi: la maggior parte dei progetti usa componenti funzionali.
- Le classi si trovano in codebase legacy: utile saperle leggere.

Confronto rapido

Stessa UI, due stili

Spiegazione

- Funzionale: più breve e moderno.
- Classe: usa this.props e this.state.
- Molti esempi moderni assumono hook.

Confronto

```
// Funzionale
function Hello({ name }) {
  return <Text>Ciao {name}</Text>;
}

// Classe
class HelloClass extends
React.Component {
  render() {
    return <Text>Ciao
{this.props.name}</Text>;
  }
}
```

Props

Input del componente

- Le props arrivano dal parent (componente sopra).
- Sono read-only: il child non deve modificarle.
- Puoi passare funzioni come callback (eventi verso l’alto).
- Tipizza le props per prevenire errori e migliorare DX.

Props con callback

Pattern fondamentale

Spiegazione

- Il parent passa una funzione al child.
- Il child la chiama quando succede un evento (es. onPress).
- Questo evita dipendenze globali e rende i componenti riutilizzabili.

Props + function

```
// Child
type PrimaryButtonProps = {
  label: string;
  onPress: () => void;
};

export function
PrimaryButton({ label, onPress }:
PrimaryButtonProps) {
  return (
    <Pressable onPress={onPress}>
      <Text>{label}</Text>
    </Pressable>
  );
}
// Parent
<PrimaryButton label="Salva"  /
```

State

Dati che cambiano

- Lo state è memoria interna (contatore, input, lista…).
- Quando lo state cambia, React aggiorna la UI (re-render).
- In funzionali: useState.
- Aggiorna in modo immutabile: crea nuove copie di array/oggetti.

useState

Contatore semplice

Spiegazione

- useState ritorna: valore + funzione setter.
- Forma funzionale: setCount(c => c+1) evita bug con valori vecchi.

useState

```
import { useState } from "react";
import { View, Text, Pressable }
from "react-native";

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

  return (
    <View>
      <Text>Conteggio:
{count}</Text>
      <Pressable onPress={() =>
setCount((c) => c + 1)}>
        <Text>+1</Text>
      </Pressable>
    </View>
  );
```

State: liste

Pattern add/toggle/remove

Spiegazione

- add: [...prev, newItem]
- toggle: map e copia dell’item
- remove: filter

Immutabilità

```
type Todo = { id: string; text:
string; done: boolean };

setTodos((prev) => [...prev, { id,
text, done: false }]);

setTodos((prev) =>
  prev.map((t) => (t.id === id ?
{ ...t, done: !t.done } : t))
);

setTodos((prev) => prev.filter((t)
=> t.id !== id));
```

Styling con StyleSheet

Come funziona

- Stili come oggetti JS (non stringhe CSS).
- StyleSheet.create aiuta con ordine e validazione.
- Unità: numeri (dp).
- Layout basato su Flexbox (column di default).

StyleSheet: esempio

Card con bordi e testo

Spiegazione

- padding/margin per spacing
- borderRadius per angoli
- fontSize/fontWeight per tipografia

StyleSheet

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

export function Card() {
  return (
    <View style={styles.card}>
      <Text style={styles.title}>Titolo</Text>
      <Text style={styles.body}>Testo
descrittivo…</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  card: {
    padding: 12,
    borderRadius: 12,
    borderWidth: 1,
    borderColor: "#e5e7eb",
    backgroundColor: "white",
  },
  title: { fontSize: 18, fontWeight: "600" },
  body: { marginTop: 6, opacity: 0.8 },
});
```

Differenze dal CSS

Cose da ricordare

- Default flexDirection = column.
- Niente selettori CSS/className (in RN).
- Shadow diverso iOS/Android (shadow* vs elevation).
- Componenti diversi: View/Text invece di div/span.

Flexbox

Le proprietà principali

- flexDirection: column/row
- justifyContent: asse principale
- alignItems: asse secondario
- flex: spazio relativo
- padding/margin: spaziatura

Flexbox: row

Layout orizzontale

Spiegazione

- row = elementi in orizzontale
- space-between distribuisce lo spazio
- alignItems centra verticalmente

Row layout

```
<View style={styles.row}>
  <Text>Sinistra</Text>
  <Text>Destra</Text>
</View>

const styles = StyleSheet.create({
  row: {
    flexDirection: "row",
    justifyContent: "space-
between",
    alignItems: "center",
    padding: 12,
  },
});
```

Liste dinamiche

ScrollView vs FlatList

- ScrollView: renderizza tutto (ok per liste piccole).
- FlatList: virtualizzazione (meglio per liste lunghe).
- FlatList richiede keyExtractor stabile.
- Usa FlatList per To-Do e casi reali.

ScrollView

Esempio semplice

Spiegazione

- Veloce per contenuti limitati.
- Non adatto a centinaia di elementi.

ScrollView

```
import { ScrollView, Text } from
"react-native";

export function SimpleList() {
  return (
    <ScrollView>
      <Text>Elemento 1</Text>
      <Text>Elemento 2</Text>
      <Text>Elemento 3</Text>
    </ScrollView>
  );
}
```

FlatList

Esempio base

Spiegazione

- data: array
- renderItem: render di ogni item
- keyExtractor: id stabile

FlatList

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

const data = [{ id: "1", label:
"Uno" }, { id: "2", label:
"Due" }];

export function BetterList() {
  return (
    <FlatList
      data={data}
      keyExtractor={(item) =>
item.id}
      renderItem={({ item }) => (
        <View style={{ padding:
12 }}>   <Text>{item.label}</Text>
        </View>
      )}
```

FlatList: consigli

Buone pratiche

- Evita index come key se l’ordine può cambiare.
- Estrai ItemSeparatorComponent per separatori.
- Per liste grandi: valuta initialNumToRender e windowSize.
- Mantieni renderItem leggero.

Hook: cosa sono

Idea centrale

- Un hook è una funzione React che aggiunge capacità ai componenti funzionali.
- Ti permette di gestire: stato, effetti, riferimenti, performance e condivisione dati.
- Regole: chiamali solo in alto (non dentro if/loop) e solo in componenti/hook.
- In React Native usi gli stessi hook di React (client).

Hook principali

Quando usarli (mappa rapida)

- useState — stato locale semplice (input, toggle, contatori).
- useEffect — side-effect (fetch, subscription, timer).
- useRef — riferimento mutabile o accesso imperativo (focus, timer).
- useCallback — stabilizzare funzioni passate a figli (liste/memo).
- useMemo — stabilizzare valori calcolati costosi (quando serve).
- useContext — dati condivisi (tema, auth, config).
- useReducer — stato complesso con transizioni prevedibili.
- useLayoutEffect — misure/layout e aggiornamenti prima del paint (casi mirati).

useEffect

Quando e perché

Spiegazione

- Esegue codice “dopo il render” (sincronizza con mondo esterno).
- Dipendenze: [] una volta, [x] quando x cambia.
- Cleanup: ritorna una funzione (unsubscribe, timer).

useEffect

```
import { useEffect, useState } from
"react";
import { Text } from "react-
native";

export function Clock() {
  const [t, setT] = useState(0);

  useEffect(() => {
    const id = setInterval(() =>
setT((v) => v + 1), 1000);
    return () => clearInterval(id);
  }, []);

  return <Text>Secondi: {t}</Text>;
}
```

useRef

A cosa serve davvero in RN

Spiegazione

- È una “scatola” che mantiene un valore tra render (non provoca re-render).
- Utile per: focus su input, scrollTo su liste, timer id, valore precedente.
- Accesso imperativo: ref.current?.metodo().

useRef

```
import { useRef } from "react";
import { TextInput, Pressable, Text
} from "react-native";

export function FocusInput() {
  const inputRef =
useRef<TextInput>(null);

  return (
    <>
      <TextInput ref={inputRef}
placeholder="Scrivi..." />
      <Pressable onPress={() =>
inputRef.current?.focus()}>
        <Text>Focus</Text>
      </Pressable>
    </>
  );
```

useCallback

Perché stabilizzare una funzione

Spiegazione

- In JS, una funzione ‘nuova’ = reference diversa ad ogni render.
- Se passi callback a figli memoizzati o righe di FlatList, può causare re-render inutili.
- useCallback “blocca” la reference finché le dipendenze non cambiano.

useCallback

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

const onPressItem =
useCallback((id: string) => {
  toggleTodo(id);
}, [toggleTodo]);

// utile se TodoRow è memoizzato o
se FlatList rende molte righe
```

useMemo

Valori calcolati (quando serve)

Spiegazione

- Memoizza un risultato (non una funzione).
- Utile solo se il calcolo è costoso o se serve stabilità per dipendenze.
- Evita ‘overuse’: prima misura, poi ottimizza.

useMemo

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

const filtered = useMemo(() => {
  return todos.filter((t) => t.done
=== showDone);
}, [todos, showDone]);
```

React.memo

Ridurre re-render nei componenti figli

Spiegazione

- React.memo memoizza un componente: se le props non cambiano, non re-renderizza.
- Molto utile per righe di lista (FlatList) quando le props sono stabili.
- Se passi funzioni ‘nuove’ ogni render, memo non aiuta (usa useCallback).

React.memo

```
import React from "react";
import { Text, Pressable } from
"react-native";

type RowProps = { label: string;
onPress: () => void };

export const Row =
React.memo(function Row({ label,
onPress }: RowProps) {
  return (
    <Pressable onPress={onPress}>
      <Text>{label}</Text>
    </Pressable>
  );
});
```

useLayoutEffect

Quando è utile in React Native

- Simile a useEffect, ma parte “prima” che lo schermo venga aggiornato (paint).
- Utile per misurazioni o aggiornamenti di layout che non devono ‘sfarfallare’.
- In RN spesso preferisci onLayout o misure native; useLayoutEffect è per casi mirati.

useLayoutEffect: esempio

Misurare layout (concetto)

Spiegazione

- Misuri dimensioni subito dopo il layout.
- Aggiorni state prima del paint (meno flicker).
- Alternativa: onLayout su View per leggere width/height.

useLayoutEffect

```
import { useLayoutEffect,
useState } from "react";
import { View } from "react-
native";

export function Box() {
  const [w, setW] = useState(0);

  useLayoutEffect(() => {
    // casi avanzati: ref + measure
(se necessario)
  }, []);

  return <View onLayout={(e) =>
setW(e.nativeEvent.layout.width)} /
>;
}
```

useContext

Condividere dati

- Evita prop drilling (passare props a catena).
- Esempi: tema, utente loggato, configurazione app.
- Crea Context + Provider, poi leggi con useContext.

useContext: esempio

Tema condiviso

Spiegazione

- Provider avvolge una parte (o tutta) l’app.
- useContext legge il valore corrente.

useContext

```
import { createContext,
useContext } from "react";
import { Text } from "react-
native";

const ThemeContext =
createContext("light");

export function Screen() {
  const theme =
useContext(ThemeContext);
  return <Text>Tema:
{theme}</Text>;
}
```

useReducer

Perché usarlo

- Quando lo state ha molte transizioni (azioni) e diventa difficile con useState.
- Quando vuoi logica prevedibile e centralizzata (stile Redux, ma locale).
- Quando più campi dipendono tra loro (form, wizard, flussi).

useReducer: anatomia

I pezzi principali

Spiegazione

- State: la situazione attuale (un oggetto).
- Action: cosa è successo (type + payload opzionale).
- Reducer: funzione pura che calcola il nuovo state.
- Dispatch: invia un’azione al reducer.

useReducer

```
import { useReducer } from "react";

type State = { count: number };
type Action =
  | { type: "inc" }
  | { type: "dec" }
  | { type: "set"; value: number };

function reducer(state: State, action: Action):
State {
  switch (action.type) {
    case "inc":
      return { count: state.count + 1 };
    case "dec":
      return { count: state.count - 1 };
    case "set":
      return { count: action.value };
  }
}

const [state, dispatch] = useReducer(reducer, {
count: 0 });
dispatch({ type: "inc" });
```

useReducer vs useState

Regola pratica

Spiegazione

- useState: perfetto per 1–3 valori indipendenti.
- useReducer: meglio quando le modifiche seguono ‘azioni’ ripetibili e combinazioni.
- Vantaggio: (state + action) ⇒ nuovo state (più testabile).

Quando scegliere

```
// useState (semplice)
const [open, setOpen] =
useState(false);

// useReducer (azioni ripetibili)
dispatch({ type: "open" });
dispatch({ type: "close" });
dispatch({ type: "toggle" });
```

useTransition

UI più fluida

Spiegazione

- Segna un update come “transition”.
- Mostra pending mentre React aggiorna in background.

useTransition

```
import { useState, useTransition }
from "react";

const [isPending, startTransition]
= useTransition();
const [query, setQuery] =
useState("");

function onChangeText(t: string) {
  startTransition(() => {
    setQuery(t);
  });
}
```

React 19 — note utili

Cosa puoi incontrare

- React 19 rafforza il concetto di Actions (async).
- Hook: useActionState, useOptimistic (molto usati sul web).
- API use() per consumare risorse/Promise in contesti supportati.
- In React Native alcuni pattern sono meno centrali, ma i concetti aiutano.

useOptimistic (React 19)

UI ottimistica (concetto)

Spiegazione

- Mostri subito un risultato “temporaneo” mentre una richiesta è in corso.
- Se la richiesta fallisce, puoi ripristinare lo state.

React 19 — useOptimistic

```
import { useOptimistic,
startTransition } from "react";

const [items, addOptimistic] =
useOptimistic(
  initialItems,
  (state, newItem) => [...state,
{ ...newItem, pending: true }]
);

function onAdd(newItem) {
  startTransition(async () => {
    addOptimistic(newItem);
    await apiCreate(newItem);
  });
}
```

useActionState (React 19)

State legato a un’azione

Spiegazione

- Aiuta a gestire: pending, errore, risultato di un’azione async.
- Molto comune in form sul web; concetto comunque utile.

React 19 — useActionState

```
import { useActionState } from
"react";

async function
saveAction(prevState, formData) {
  return { ok: true, message:
"Salvato" };
}

const [state, action, isPending] =
useActionState(saveAction, {
  ok: false,
  message: "",
});
```

Pausa

10-15min

Esercizio pratico

To-Do List App

- Creiamo una schermata con task.
- Aggiungi task da input.
- Segna completato toccando il task.
- Rimuovi task con azione dedicata.
- Render con FlatList (key stabile).

To-Do: modello dati

Tipo Todo

- Todo = { id, text, done }
- id serve per key e per identificare item
- done indica completato/non completato

To-Do: state iniziale

todos + inputText

Spiegazione

- todos: array nello state
- inputText: testo dell’input
- lista iniziale vuota

TodoScreen

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

type Todo = { id: string; text:
string; done: boolean };

export function TodoScreen() {
  const [todos, setTodos] =
useState<Todo[]>([]);
  const [inputText, setInputText] =
useState("");
}
```

To-Do: aggiunta task

TextInput controllato + addTodo

Spiegazione

- trim per evitare task vuoti
- id semplice con Date.now()
- reset input dopo add

Add Todo

```
import { View, TextInput, Pressable, Text } from "react-
native";

function addTodo() {
  const text = inputText.trim();
  if (!text) return;

  const id = String(Date.now());
  setTodos((prev) => [...prev, { id, text, done:
false }]);
  setInputText("");
}

return (
  <View style={{ flexDirection: "row", gap: 8 }}>
    <TextInput
      value={inputText}
      onChangeText={setInputText}
      placeholder="Scrivi un task…"
      style={{ flex: 1, borderWidth: 1, padding: 10,
borderRadius: 10 }}
    />
    <Pressable
      onPress={addTodo}
      style={{ padding: 10, borderRadius: 10,
backgroundColor: "#111827" }}
      <Text style={{ color: "white" }}>Aggiungi</Text>
    </Pressable>
  </View>
```

To-Do: lista

FlatList + TodoRow

Spiegazione

- FlatList renderizza la lista
- TodoRow incapsula UI della riga
- Passi callback onToggle/onRemove

Render list

```
import { FlatList } from "react-
native";

<FlatList
  data={todos}
  keyExtractor={(t) => t.id}
  renderItem={({ item }) => (
    <TodoRow todo={item}
onToggle={toggleTodo}
onRemove={removeTodo} />
  )}
/>
```

To-Do: TodoRow

Componente riga con props

Spiegazione

- Pressable sul testo per toggle
- Bottone ‘Elimina’ per rimozione
- Stile done con line-through

TodoRow

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

type Todo = { id: string; text: string; done: boolean };

type TodoRowProps = {
  todo: Todo;
  onToggle: (id: string) => void;
  onRemove: (id: string) => void;
};

export function TodoRow({ todo, onToggle, onRemove }:
TodoRowProps) {
  return (
    <View style={{ flexDirection: "row", alignItems:
"center", padding: 12 }}>
      <Pressable onPress={() => onToggle(todo.id)}
style={{ flex: 1 }}>
        <Text style={{ textDecorationLine: todo.done ?
"line-through" : "none" }}>
          {todo.text}
        </Text>
      </Pressable>

      <Pressable onPress={() => onRemove(todo.id)}>
        <Text style={{ color: "#ef4444", fontWeight:
"600" }}>Elimina</Text>
      </Pressable>
    </View>
  );
```

To-Do: logica

toggleTodo e removeTodo

Spiegazione

- toggle usa map e copia dell’item
- remove usa filter
- immutabilità sempre

Logic

```
function toggleTodo(id: string) {
  setTodos((prev) =>
    prev.map((t) => (t.id === id ?
{ ...t, done: !t.done } : t))
  );
}

function removeTodo(id: string) {
  setTodos((prev) =>
prev.filter((t) => t.id !== id));
}
```

Ricapitolando

Cosa hai imparato

- JSX: UI + espressioni con { }.
- Componenti: composizione e riuso.
- Props: dati e callback.
- State: useState e update immutabili.
- StyleSheet/Flexbox: layout e stile.
- Liste: ScrollView vs FlatList.
- Hook: principali + useReducer/useLayoutEffect/React.memo + note React 19.

Domande di ripasso

Domanda 1 di 5

1. Quale affermazione su JSX è corretta?

- ☐ JSX è HTML e viene eseguito direttamente dal browser
- ☐ JSX è una sintassi che viene trasformata in chiamate JavaScript
- ☐ JSX può essere usato solo con componenti a classe
- ☐ JSX sostituisce completamente JavaScript

Domande di ripasso

Domanda 2 di 5

2. In React Native, quale coppia di componenti è tipica per layout e testo?

- ☐ div e span
- ☐ section e p
- ☐ View e Text
- ☐ table e tr

Domande di ripasso

Domanda 3 di 5

3. Qual è la regola corretta per aggiornare uno state array?

- ☐ Modificare l’array esistente con push()
- ☐ Creare un nuovo array (es. con spread, map, filter)
- ☐ Usare sempre variabili globali
- ☐ Aggiornare lo state dentro un loop while

Domande di ripasso

Domanda 4 di 5

4. Quando preferisci FlatList rispetto a ScrollView?

- ☐ Quando la lista può essere lunga e vuoi virtualizzazione/performance
- ☐ Quando hai massimo 3 elementi
- ☐ Quando vuoi disabilitare lo scroll
- ☐ Quando vuoi usare CSS className

Domande di ripasso

Domanda 5 di 5

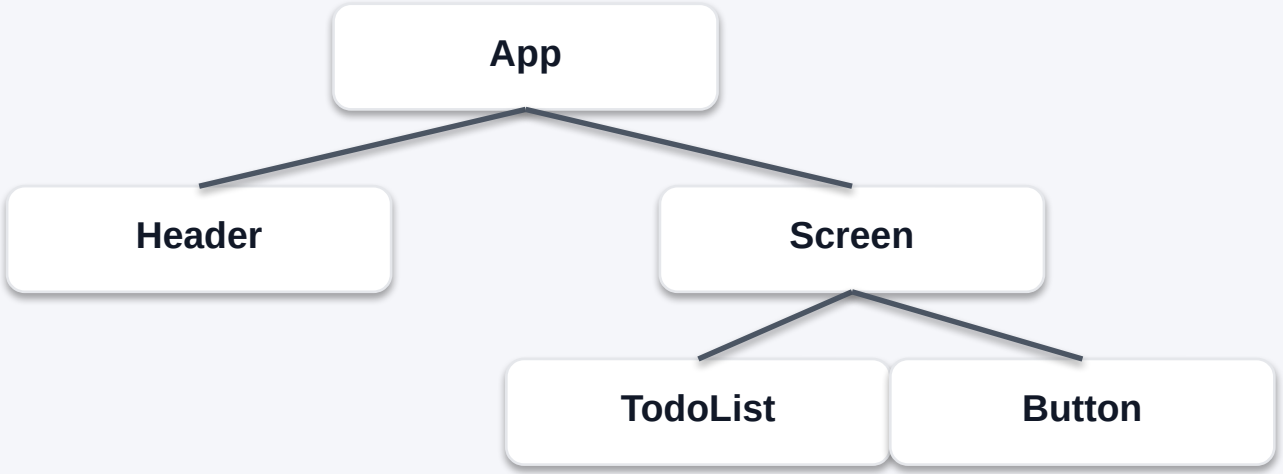
5. Quale opzione descrive meglio i componenti React Native?

- ☐ Sono file CSS che descrivono lo stile dell’app
- ☐ Sono funzioni/classi che restituiscono UI e possono ricevere props
- ☐ Sono solo template statici senza logica
- ☐ Sono librerie native scritte solo in Swift/Kotlin

Update → re-render → UI

Cosa succede quando cambia lo stato

Esempio concettuale di “tree” (componenti annidati)



- Un update (setState/dispatch) chiede a React di ricalcolare la UI.
- React “rende” il tree: ricrea la descrizione della UI (non disegna ancora).
- Poi confronta (diff) con il tree precedente e capisce cosa è cambiato

Avvio React Native CLI

Comandi e toolchain (Android + iOS)

- Obiettivo: avviare l'app su device/emulatore usando Metro + build nativa.
- Servono due pezzi: (1) Metro bundler, (2) toolchain nativa Android/iOS.
- La UI la modifichi sempre in App.tsx (hot reload/fast refresh).

```
# terminale 1 (sempre acceso)
npx react-native start

# terminale 2
npx react-native run-android
# oppure
npx react-native run-ios
```

Workflow consigliato

Due terminali, IDE opzionali

- Metro può restare in un terminale dedicato (porta 8081).
- Android Studio / Xcode non devono rimanere aperti: basta emulator/simulator acceso.
- Se cambi App.tsx e salvi: Fast Refresh aggiorna su device.

```
# Android (emulatore già avviato) npx
react-native run-android

# iOS (simulator già avviato) npx react-
native run-ios

# log utili
npx react-native log-android npx
react-native log-ios

# crea app react-native-CLI bare
npx @react-native-community/cli@
latest init MyApp
```

Metro

Bundler e dev server

- Metro trasforma JS/TS e serve il bundle all'app in debug.
- Porta tipica: http://localhost:8081 (non è una UI web).
- Comandi Metro: r reload, d Dev Menu, j DevTools.

```
# Metro
npx react-native start

# se il device non carica il bundle:
adb reverse tcp:8081 tcp:8081

# dev menu (Android)
adb shell input keyevent 82
```

Android build (Gradle)

Cosa sta succedendo quando fai run-android

- run-android lancia Gradle: compila, installa l'APK debug e avvia l'app.
- Durante la prima build può installare componenti SDK (Build-Tools, NDK, CMake).
- Pulizia in caso di problemi strani: gradlew clean + restart Metro.

```
cd android
./gradlew clean
cd ..

# debug install
npx react-native run-android

# release (AAB per Play Store)
cd android
./gradlew bundleRelease
```

Java / JDK (Temurin)

Errore tipico: Unable to locate a Java Runtime

- Gradle richiede un JDK installato e visibile a macOS.
- Su Homebrew: temurin@17 è la scelta piu compatibile per build Android.
- Verifica: java -version e JAVA_HOME.

```
brew install --cask temurin@17

export JAVA_HOME=$(/usr/libexec/java_home -v 17)
export PATH="$JAVA_HOME/bin:$PATH"

java -version
echo $JAVA_HOME
```

Device Android vs Emulatore

Connessione a Metro

- Telefono via USB: spesso serve adb reverse per esporre 8081 al device.
- Emulatore: di solito funziona senza reverse; in caso di errore imposta host Metro.

```
# lista device collegati
adb devices

# USB device
adb reverse tcp:8081 tcp:8081
```

iOS: CocoaPods

Errore xcodebuild 65 / file Pods mancanti

- Se mancano ios/Pods/*xcconfig o *xcfilelist: non hai fatto pod install.
- Quando apri in Xcode usa sempre la .xcworkspace (non .xcodeproj).
- Dopo nuove librerie native: pod install e rebuild.

```
cd ios
pod install
cd ..

# pulizia forte se necessario
cd ios
rm -rf Pods Podfile.lock
pod install
cd ..
```