Make Your App Not Feel Vibe-Coded

The four features from the Tiles Day 10 video, and how to build each one in an Expo / React Native app.

Everything here is lifted from how I built Tiles (an Expo SDK 57 app on Supabase). The code snippets are trimmed from the real files, so they work. Where something is a trap, I say so.

README · BEFORE YOU START

- You need a **development build** (`npx expo run:ios` or EAS). None of the native pieces (Action Button, widgets) work in Expo Go.
- You need a paid Apple Developer account for the App Group the widget and the Action Button share.
- The Action Button needs an **iPhone 15 Pro or later**. Widgets work on any iPhone.
- Install Expo packages with `npx expo install`, never plain `npm install`.

#	FEATURE	WHY IT MAKES THE APP FEEL REAL	EFFORT
1	**Action Button utility**	Your app does something *without opening*.	Hard
2	**Home-screen widget**	Your app lives on the home screen.	Medium
3	**Animated onboarding**	Users see the value before they sign up.	Medium
4	**Micro-interactions**	Every tap feels answered.	Easy

Build in the order 4 → 3 → 2 → 1 if you want quick wins first. The video order is 1 → 4 for impact.

01 · FEATURE ONE

1. Action Button `utility`

What it does in Tiles: press the Action Button, and a prompt drops out of the Dynamic Island asking *Which account? Which category? How much?* The spend is saved. Tiles never opens.

What iOS actually lets you do

You can't program the Action Button or draw your own UI in the Dynamic Island. What you *can* do is expose an **App Intent** as an **App Shortcut**. The user then picks it in **Settings** → **Action Button** → **Shortcut** → ***Your App***. iOS draws the prompts itself.

There is a second route, a **Control** (iOS 18+), but a control runs with **no UI**: it can only open your app. Tiles offers both: the Shortcut for the in-place prompt, the Control as a fallback that opens the Add Spend screen via a deep link (`tiles://add-spend`).

The architecture

The intent runs **outside your JavaScript app**. So it can't call your store. Split the job in three:

```
App (JS)  ──writes accounts + categories──▶  App Group (UserDefaults)
Intent (Swift) ◀──reads them to build the prompt──┘
Intent (Swift) ──appends the entry──▶  App Group "pending" queue
App (JS)  ◀──on launch / becoming active, drains the queue and saves──┘
```

The App Group is a shared `UserDefaults` that both the app and the intent can read.

Step 1: share data from JS

COPY-PASTE · TERMINAL

```
npx expo install @bacons/apple-targets
```

Use only its `ExtensionStorage` module (see the trap below).

```typescript
import { ExtensionStorage } from '@bacons/apple-targets';

const APP_GROUP = 'group.com.yourname.yourapp'; // must match the Swift side

// Debounce this, and only write when the data actually changed.
new ExtensionStorage(APP_GROUP).set('quickEntry.accounts', JSON.stringify(accounts));
new ExtensionStorage(APP_GROUP).set('quickEntry.categories', JSON.stringify(categories));
```

Share **only what the prompt needs** (id, name, currency). Never balances.

Step 2: the App Intent (Swift)

Make each prompt a **required** `@Parameter`, declared in the order you want them asked. iOS then asks for each one itself before `perform` runs.

```swift
import AppIntents

struct AddSpendIntent: AppIntent {
  static let title: LocalizedStringResource = "Add Spend"
  static let openAppWhenRun = false          // this is what keeps your app closed

  @Parameter(title: "Account",  requestValueDialog: IntentDialog("Which account?"))
  var account: AccountEntity                 // an AppEntity backed by the shared list

  @Parameter(title: "Category", requestValueDialog: IntentDialog("Which category?"))
  var category: CategoryEntity

  @Parameter(title: "Amount",   requestValueDialog: IntentDialog("How much?"))
  var amount: Double

  func perform() async throws -> some IntentResult & ProvidesDialog {
    guard amount.isFinite, amount > 0 else {
      throw $amount.needsValueError(IntentDialog("Enter an amount above zero."))
    }
    Store.appendPending(accountID: account.id, amount: amount, category: category.id)
    return .result(dialog: "Spent \(amount) on \(category.name). It shows in the app when you next
open it.")
  }
}
```

`AccountEntity` and `CategoryEntity` conform to `AppEntity` with an `EntityStringQuery` that reads the shared JSON from the App Group. Return a clear error ("Open the app and add an account first") instead of an empty list when nothing has been shared yet.

TRAP

Don't make the parameters optional and ask with `requestValue` inside `perform`. From the Action Button that showed only a "+" and no prompt.

Step 3: make it appear in the Action Button list

An intent alone is just an action for building shortcuts. To be listed under *Action Button → Shortcut → Your App* with no setup, add an `AppShortcutsProvider` **in the app target** (an app can have only one):

SWIFT

```
struct AppShortcuts: AppShortcutsProvider {
  static var appShortcuts: [AppShortcut] {
    AppShortcut(
      intent: AddSpendIntent(),
      phrases: ["Add a spend in \(.applicationName)"],
      shortTitle: "Add Spend",
      systemImageName: "creditcard"
    )
  }
}
```

Step 4: record the entry in JS

When the app mounts and each time it becomes active, read the queue, save each entry through your normal store, then write back only the entries you haven't handled (the intent may have queued another in the meantime):

TYPESCRIPT

```
useEffect(() => {
  const drain = () => applyPendingEntries(readPending(), store);
  drain();
  const sub = AppState.addEventListener('change', (s) => s === 'active' && drain());
  return () => sub.remove();
}, []);
```

Date the entry with the time the button was pressed (`createdAt` from the intent), not the time you drained it. Send the amount as a **string**, not a float.

01 · ACTION BUTTON UTILITY · CONTINUED

Step 5: get the Swift files into the Xcode project

Expo regenerates `ios/` on prebuild, so don't edit it by hand. Keep your Swift in a folder like `native/quick-entry/` and copy it in with a **config plugin** (`withXcodeProject` + `withDangerousMod`) that adds the files to the app target. Tiles also appends its control to the widget extension that `expo-widgets` generates (see section 2).

TRAPS

- Don't add `@bacons/apple-targets` to your `plugins` array if you also use `expo-widgets`. With no target of its own name it "updates" the widget target and prebuild fails. Import only the `ExtensionStorage` module.
- Your plugin must run **after** `expo-widgets`, which recreates its target folder. Wrap the whole config so your mod runs last.
- After editing Swift, re-run `npx expo prebuild -p ios` (or `npm run ios`). A JS reload won't pick it up.
- **If your app isn't listed:** rebuild, open the app once, then restart the phone. iOS caches the shortcut list.

Step 6: teach users to turn it on

Apple has no deep link to the Action Button page. Tiles' guide screen has an **Open Settings** button (`Linking.openSettings()`) and five numbered steps. Show the feature in your onboarding too (see section 3).

COPY-PASTE · PROMPT FOR YOUR CODING AGENT

"Add an iOS Action Button utility to my Expo app. Create an App Intent with required Account, Category and Amount parameters, an `AppShortcutsProvider`, and an App Group queue. The app shares accounts via `ExtensionStorage`, and drains the queue on launch and when it becomes active. Add a config plugin that copies the Swift files into the Xcode project after `expo-widgets`."

02 · FEATURE TWO

2. Home-screen `widget`

What it does in Tiles: a large widget showing the net-worth mosaic, on a clear background so the wallpaper shows through. Tapping it opens the app.

Step 1: install and configure

COPY-PASTE · TERMINAL

```
npx expo install expo-widgets @expo/ui
```

APP.CONFIG.TS

```
// app.config.ts
plugins: [
  [
    'expo-widgets',
    {
      widgets: [
        {
          name: 'MosaicWidget',
          displayName: 'Mosaic',
          description: 'What you own and what you owe at a glance.',
          ios: { supportedFamilies: ['systemLarge'], contentMarginsDisabled: true },
        },
      ],
    },
  ],
],
```

The App Group defaults to `group.<your bundle id>`.

Step 2: draw it with the `'widget'` directive

The widget runs in the extension's **own JavaScript runtime**, not your app's. That has hard rules:

- Mark the component with `"widget"`.
- Use only `@expo/ui/swift-ui` views. No hooks.
- Declare **everything the widget uses inside the function**. It can't import your helpers at runtime.

TSX

```tsx
import { Text, VStack, ZStack } from '@expo/ui/swift-ui';
import { containerBackground, frame, offset, widgetURL } from '@expo/ui/swift-ui/modifiers';
import { createWidget, type WidgetEnvironment } from 'expo-widgets';

const MosaicWidget = (props: { tiles: Tile[] }, environment: WidgetEnvironment) => {
  'widget';
  const dark = environment.colorScheme === 'dark';
  // ...draw each tile in a ZStack with frame() + offset(), as plain data from props
  return (
    <ZStack modifiers={[containerBackground('clear', 'widget'), widgetURL('myapp://')]}>
      {/* tiles */}
    </ZStack>
  );
};

export default createWidget('MosaicWidget', MosaicWidget);
```

Step 3: do the thinking in the app, not the widget

A widget can't measure itself and can't run your shaders. So:

- **Pre-compute the layout in the app** and send the widget finished data (position, size, colour, preformatted text such as `£12,400` or `••••`).
- Look up the widget's size from Apple's widget-size table for the phone's screen size.
- Replace anything fancy with a SwiftUI gradient. Tiles' app uses a Skia metal shader; the widget gets a `linearGradient` approximation worked out in the app.

02 · HOME-SCREEN WIDGET · CONTINUED

Step 4: keep it fresh

Mount one sync component near the root. Debounce (500 ms), skip identical props, and never let a widget failure crash the app:

TYPESCRIPT

```
function send(pending) {
  if (!pending.latest || jsonEqual(pending.latest, pending.sent)) return;
  try {
    MosaicWidget.updateSnapshot(pending.latest); // each write reloads the widget
    pending.sent = pending.latest;
  } catch (e) {
    if (__DEV__) console.warn('Widget update failed', e);
  }
}
```

Also write when the app goes to the background. Respect your **hide-values** setting in the widget text.

TRAPS

- Android: `expo-widgets` is iOS only. Guard with `Platform.OS === 'ios'`.
- The widget only changes when the app has run. Tell users that, or it looks broken.
- Widgets can't animate. Keep them still and crisp.

COPY-PASTE · PROMPT FOR YOUR CODING AGENT

"Add an `expo-widgets` home-screen widget (systemLarge, clear background) that shows my top accounts. Use the `'widget'` directive and only `@expo/ui/swift-ui` views, with everything declared inside the function. Compute layout and formatted text in the app, send it with `updateSnapshot` from a debounced sync component, and open the app on tap with `widgetURL`."

03 · FEATURE THREE

3. Animated onboarding

The idea: don't describe the features with static screenshots. Show **mock UI that moves**, built from your real components, and finish with a short video demo.

Pattern A: one clock drives everything

Each Tiles demo page (the Action Button page is a 10-second loop) runs on **one timeline**. Every animated thing derives from a single time value, so it's all in sync and trivial to retime.

ACTION-BUTTON-TIMELINE.TS

```
// action-button-timeline.ts: milliseconds from the start of each pass
export const AB = {
  press:       [600, 1_100],   // [start, length]
  drop:        [1_800, 600],   // prompt drops out of the Dynamic Island
  tapAccount:  [3_200, 300],
  toCategory:  [3_500, 320],
  tapCategory: [4_500, 300],
  type:        [5_500, 900],   // the amount is typed key by key
  toConfirm:   [7_100, 320],
  close:       [9_000, 500],
  loop: 10_000,
} as const;
```

Then pure functions turn time into state: `actionButtonPhase(t)`, `pressDepth(t)`, `typedAt('12.5', t)`. Mark them `'worklet'` so Reanimated can run them on the UI thread. Pure functions are also easy to unit-test.

TSX

```
const t = useSharedValue(0);
useEffect(() => {
  t.value = withRepeat(withTiming(AB.loop, { duration: AB.loop, easing: Easing.linear }), -1);
}, []);
const buttonStyle = useAnimatedStyle(() => ({ transform: [{ translateX: -4 * pressDepth(t.value) }]
}));
```

03 · ANIMATED ONBOARDING · CONTINUED

Pattern B: the phone frame

Draw a `PhoneFrame` in code (rounded rect, Dynamic Island, wallpaper), and put your mock UI inside it. No screenshots to re-export when your design changes.

Pattern C: copy the real thing exactly

For the Action Button page, copy the **real intent's wording** ("Which account?", "How much?", the confirmation). Then add a test that reads your Swift file and fails if the wording changes. Your mock and your product can't drift apart.

Pattern D: a video for the hand-off

For the final screen, loop a short screen recording with `expo-video`, muted, over a poster frame:

COPY-PASTE · TERMINAL

```
npx expo install expo-video
```

TSX

```
const player = useVideoPlayer(require('@/assets/demo-light.mp4'), (p) => { p.loop = true; p.muted =
true; p.play(); });
return <VideoView player={player} nativeControls={false} style={styles.video} />;
```

Ship light and dark versions and pick by `useColorScheme()`.

03 · ANIMATED ONBOARDING · CONTINUED

Rules that keep it feeling polished

1. **Play once or loop deliberately.** Static facts (privacy) play once; processes (the Action Button) loop.
2. **Stagger.** Tiles settle in about 90 ms apart, never all at once.
3. **Honour Reduce Motion.** `useReducedMotion()` from Reanimated; fall back to a fade or the end state.
4. **One idea per page.** Title, one line of subtitle, one animation.
5. **Skip pages that don't apply.** Tiles shows the Action Button and widget pages on iPhone only.
6. **Track the funnel.** Log a step-completed event per page so you can see where people leave.

COPY-PASTE · PROMPT FOR YOUR CODING AGENT

"Build an onboarding page with an animated mock phone. Define a timeline constant of `[start, length]` beats and a loop length, derive every style from one Reanimated shared value `t` with `'worklet'` pure functions, honour Reduce Motion, and unit-test the phase functions."

04 · FEATURE FOUR

4. Micro-interactions

Small, consistent, everywhere. This is what separates "works" from "feels made".

Rule: one shared component, not 40 ad-hoc ones

Tiles has a single `PressableScale` used by tiles, cards and rows. Press scales to 0.97 in 120 ms, release returns in 200 ms, optional haptic:

TSX · PRESSABLESCALE

```tsx
import * as Haptics from 'expo-haptics';
import Animated, { useAnimatedStyle, useReducedMotion, useSharedValue, withTiming } from 'react-native-reanimated';

const AnimatedPressable = Animated.createAnimatedComponent(Pressable);

export function PressableScale({ children, haptic, onPress, ...rest }) {
  const reduceMotion = useReducedMotion();
  const scale = useSharedValue(1);
  const style = useAnimatedStyle(() => ({ transform: [{ scale: scale.value }] }));
  return (
    <AnimatedPressable
      accessibilityRole="button"
      {...rest}
      onPressIn={() => { if (!reduceMotion) scale.value = withTiming(0.97, { duration: 120 }); }}
      onPressOut={() => { scale.value = withTiming(1, { duration: 200 }); }}
      onPress={(e) => { if (haptic) void Haptics.selectionAsync(); onPress?.(e); }}
      style={[rest.style, style]}
    >
      {children}
    </AnimatedPressable>
  );
}
```

The checklist

DETAIL	WHAT TILES DOES
Buttons	Scale to 0.97 on press-in; a light haptic on primary actions only.
Haptics	`selectionAsync` for taps, success haptic when something completes (e.g. a share). Don't buzz on everything.
Screen transitions	Root tabs *slide by position*: moving right slides content left. 320 ms ease-out cubic. Falls back to a fade under Reduce Motion.
Modals	Real native page sheets on iOS (drag to dismiss), not custom overlays.
Keyboard	Pages and buttons ride up with the keyboard (`useReanimatedKeyboardAnimation`).
Hide / reveal values	See below.
Latency	Trigger on `onPressIn` (touch-down), not `onPress` . It feels instant.

The hide-values animation (the one people notice)

Sensitive numbers shouldn't just swap to `••••` . In Tiles each character becomes its own animated glyph:

- **Hiding:** digits collapse **right to left**, then four dots grow in **left to right**. About 0.4–0.55 s total, with the stagger shortened for long amounts.
- **Showing:** the same timeline, reversed.
- All glyph sizes are **measured ahead of time and cached**, so the first frame moves immediately.
- Only animate within ~300 ms of the setting flipping. A tab that unfreezes later, or a sheet opened later, should show the final state, not replay.
- Funnel *all* money display through one component (`MaskedAmount`) so the behaviour is identical everywhere, including the widget and share image.

Make the details consistent

Put sizes in one place (`Metrics` : gutter 16, button height 52, card radius 22) and animations in a handful of shared components. If every button, sheet and amount goes through a shared component, polish is a one-line change.

COPY-PASTE · PROMPT FOR YOUR CODING AGENT

"Create a `PressableScale` component using Reanimated (scale to 0.97 on press-in, 200 ms back, `expo-haptics` selection on press when `haptic` is set, no scale under Reduce Motion) and replace every `TouchableOpacity` and `Pressable` card/row/tile in the app with it."

05 · BEFORE YOU SHIP

Ship checklist

- [ ] Dev build installs on a real iPhone (Action Button and widget don't work in Expo Go).
- [ ] App Group id matches in Swift and JS.
- [ ] Action Button: app listed under Settings → Action Button → Shortcut; entry appears after opening the app.
- [ ] Widget: added to the home screen, updates after an entry, respects hide-values.
- [ ] Onboarding pages for iPhone-only features are skipped on Android and iPad.
- [ ] Reduce Motion tested on every animation.
- [ ] Android doesn't crash: native-only code is behind `Platform.OS === 'ios'`.
- [ ] Haptics only on meaningful actions.

Dependencies used

`expo-widgets`, `@expo/ui`, `@bacons/apple-targets` (storage module only), `react-native-reanimated`, `expo-haptics`, `expo-video`, and `@shopify/react-native-skia` (for Tiles' metal shader; optional for you).