WaterMark Camera App - Complete Learning Guide

Table of Contents

High-Level Overview

What is WaterMark?

WaterMark is a **premium camera application** built with React Native and Expo that allows users to capture photos with a visually stunning, award-worthy UI featuring: - **Modern animated controls** with rotating gradients and spring physics - **Full-screen photo viewing** with gesture-based controls - **Photo gallery management** with delete and share functionality - **Dark-blue-purple-pink theme** for a premium aesthetic

Key Features

1. **Camera Capture** - Real-time camera preview with flash and torch controls
2. **Animated UI** - 60fps animations using Reanimated v4
3. **Photo Management** - Save, view, delete, and share photos
4. **Gallery View** - Grid-based photo browsing with selection
5. **Full-Screen Viewer** - Immersive photo viewing experience

---

Project Architecture

Directory Structure

```
WaterMark/
  app/                          # Expo Router screens
     _layout.tsx               # Root navigation layout
     index.tsx                 # Main camera screen
  src/
```

```
    components/              # Reusable UI components
        CaptureButton.tsx    # Animated photo capture button
        PhotoThumbnail.tsx   # Latest photo preview
        GalleryModal.tsx     # Photo grid modal
        PhotoViewer.tsx      # Full-screen photo viewer
        PermissionPage.tsx   # Camera permission UI
        NoCameraDeviceError.tsx
    constants/
        Colors.ts            # App-wide color theme
    utils/
         PhotoManager.ts      # Photo storage logic
  assets/                      # Images and static files
  app.json                     # Expo configuration
  package.json                 # Dependencies
```

Architecture Pattern: Component-Based UI with Hooks

The app follows React's component-based architecture with functional components and hooks:

```
        app/index.tsx (Root)
  - Camera initialization
  - State management (photos, UI)
  - Event handlers
```

```
  Camera          UI Layer
  Layer           Components
```

```
  Capture         Gallery          Photo
  Button          Modal            Viewer
```

Core Technologies

1. **React Native (v0.81.5)**
   - **What**: Cross-platform mobile framework

- **Why**: Write once, run on iOS and Android
- **Key Concepts**:
  - Components render native UI elements
  - StyleSheet creates optimized styles
  - Hooks manage component lifecycle and state

2. **Expo (v54)**

- **What**: Development platform for React Native
- **Why**: Simplifies native module integration
- **Features Used**:
  - `expo-router`: File-based navigation
  - `expo-camera`: Camera access (via vision-camera)
  - `expo-haptics`: Tactile feedback
  - `expo-linear-gradient`: Gradient rendering

3. **React Native Reanimated (v4)**

- **What**: High-performance animation library
- **Why**: Runs animations on UI thread (60fps guaranteed)
- **Key Features**:
  - **Shared Values**: Animated values that persist across renders
  - **Worklets**: JavaScript functions that run on UI thread
  - **Spring Physics**: Natural, realistic motion

4. **React Native Vision Camera (v4.7.3)**

- **What**: Advanced camera library
- **Why**: Better performance and features than Expo Camera
- **Capabilities**:
  - Fast photo/video capture
  - Flash and torch control
  - Multiple device selection

5. **TypeScript**

- **What**: JavaScript with static typing
- **Why**: Catch errors at compile-time, better IDE support
- **Benefits**:
  - Autocomplete for props and functions
  - Type safety prevents runtime errors
  - Self-documenting code

Component Deep Dive

Component 1: `CaptureButton.tsx`

Purpose The main photo capture button with a **rotating gradient ring** and spring-based press animations.

Architecture

```
Component Structure:
  Shared Values (rotation, scale, innerScale, progress)
  Effects (rotation animation, capture progress)
  Event Handlers (pressIn, pressOut, press)
  Animated Styles (outerRing, innerCircle, progress)
  Render (Pressable > AnimatedViews > LinearGradient)
```

How It Works 1. Shared Values Setup

```
const rotation = useSharedValue(0);
const scale = useSharedValue(1);
const innerScale = useSharedValue(1);
const progress = useSharedValue(0);
```

- `useSharedValue`: Creates animated values that live on UI thread
- `rotation`: 0-360 degrees for ring rotation
- `scale`: Outer ring size (1 = normal, 1.15 = pressed)
- `innerScale`: Inner circle size (0.85 when pressed)
- `progress`: Capture progress indicator (0-1)

2. Continuous Rotation Animation

```
useEffect(() => {
  rotation.value = withRepeat(
    withTiming(360, {
      duration: 10000,
      easing: Easing.linear,
    }),
    -1,  // Infinite repeat
    false
  );
}, []);
```

- `withRepeat`: Loops animation indefinitely
- `withTiming`: Animates value over time
- Takes 10 seconds for full 360° rotation
- Runs continuously in background

3. Press Handler with Spring Physics

```tsx
const handlePressIn = () => {
  Haptics.impactAsync(Haptics.ImpactFeedbackStyle.Medium);
  scale.value = withSpring(1.15, {
    damping: 10,    // Controls bounce
    stiffness: 300, // Controls speed
  });
  innerScale.value = withSpring(0.85);
};
```

- **Haptic Feedback**: Physical vibration on press
- **Spring Animation**: Natural, bouncy motion
  - `damping`: Lower = more bounce (10 is bouncy)
  - `stiffness`: Higher = faster animation (300 is snappy)
- Outer ring expands to 1.15x while inner shrinks to 0.85x

4. Animated Style Computation

```tsx
const outerRingStyle = useAnimatedStyle(() => ({
  transform: [
    { rotate: `${rotation.value}deg` },
    { scale: scale.value },
  ],
  opacity: disabled ? 0.5 : 1,
}));
```

- `useAnimatedStyle`: Creates styles that update on UI thread
- **Worklet**: This function runs 60 times per second without blocking JS thread
- Combines rotation and scale transformations

5. Gradient Ring

```tsx
<LinearGradient
  colors={Colors.captureRingGradient as any}
  start={{ x: 0, y: 0 }}
  end={{ x: 1, y: 1 }}
  style={styles.gradientRing}
/>
```

- Uses color array: Pink → Purple → Blue → Cyan → Pink
- Diagonal gradient (top-left to bottom-right)
- Rotates with the outer ring

Key Learning Points

- **UI Thread Animations**: All animations run on dedicated thread for 60fps
- **Spring Physics**: More natural than linear timing
- **Haptic Feedback**: Enhances user interaction feel

- **Gradient Rotation**: Visual interest without complex graphics

---

Component 2: `PhotoThumbnail.tsx`

Purpose Displays the most recently captured photo with entrance and update animations.

Unique Features

- **Slide-in entrance**: First photo slides from left
- **3D flip animation**: Flips when new photo replaces old one
- **Shimmer loading**: Shows loading state during capture
- **State tracking**: Uses `useState` and `useRef` to avoid Reanimated warnings

How It Works 1. State Management

```
const [hasAppeared, setHasAppeared] = useState(false);
const previousPhotoUri = useRef<string | null>(null);
```

- `hasAppeared`: Tracks if thumbnail has shown at least once
- `previousPhotoUri`: Stores last photo URI to detect changes
- **Why useRef?** Doesn't trigger re-renders, persists across renders

2. Entrance Animation Logic

```
useEffect(() => {
  if (photoUri && !hasAppeared) {
    opacity.value = withTiming(1, { duration: 300 });
    translateX.value = withSpring(0, {
      damping: 15,
      stiffness: 200,
    });
    setHasAppeared(true);
  }
}, [photoUri, hasAppeared]);
```

- **Conditional**: Only runs on first photo
- **Opacity**: Fades in from 0 to 1
- **TranslateX**: Slides from -50px to 0
- **setHasAppeared**: Prevents re-running on updates

3. 3D Flip Animation

```
useEffect(() => {
  if (photoUri && hasAppeared &&
      previousPhotoUri.current !== null &&
      previousPhotoUri.current !== photoUri) {
```

```
    rotateY.value = withSequence(
      withTiming(90, { duration: 150 }),  // Flip to edge
      withTiming(0, { duration: 150 })     // Flip back
    );
  }
  previousPhotoUri.current = photoUri;
}, [photoUri, hasAppeared]);
```

- **withSequence**: Chains animations back-to-back
- **Step 1**: Rotate to 90° (edge view) in 150ms
- **Step 2**: Rotate back to 0° (new photo shows) in 150ms
- **Total**: 300ms smooth flip effect

4. Shimmer Loading Effect

```
useEffect(() => {
  if (isLoading) {
    shimmerTranslate.value = withRepeat(
      withTiming(100, {
        duration: 1500,
        easing: Easing.linear,
      }),
      -1,
      false
    );
  }
}, [isLoading]);
```

- Moves gradient from -100 to 100 (left to right)
- Repeats infinitely while `isLoading` is true
- Creates "scanning" shimmer effect

Key Learning Points

- **useState vs useRef**: Choose based on re-render needs
- **useEffect Dependencies**: Control when effects run
- **withSequence**: Chain multiple animations
- **Conditional Animations**: Run different animations based on state

---

Component 3: `GalleryModal.tsx`

Purpose Full-screen modal displaying all captured photos in a 3-column grid with selection and actions.

Architecture

`GalleryModal`

```
Modal Container (slide-up animation)
Backdrop (tap to close)
Header (title + close button)
FlatList (photo grid)
    PhotoGridItem × N
        Image
        Selection overlay (if selected)
        Press animations
Action Bar (delete/share, if photo selected)
```

How It Works 1. Modal Slide Animation

```
const translateY = useSharedValue(SCREEN_HEIGHT);
const backdropOpacity = useSharedValue(0);

useEffect(() => {
  if (visible) {
    translateY.value = withSpring(0, {
      damping: 30,
      stiffness: 300,
    });
    backdropOpacity.value = withTiming(1, { duration: 300 });
  } else {
    translateY.value = withTiming(SCREEN_HEIGHT, { duration: 250 });
    backdropOpacity.value = withTiming(0, { duration: 250 });
  }
}, [visible]);
```

- **Initial State**: Modal is off-screen (Y = SCREEN_HEIGHT)
- **Open**: Springs up to Y = 0
- **Close**: Slides down with timing animation
- **Backdrop**: Fades in/out simultaneously

2. Photo Grid with FlatList

```
<FlatList
  data={photos}
  renderItem={renderPhoto}
  keyExtractor={(item) => item.id}
  numColumns={3}
  contentContainerStyle={styles.gridContent}
  ListEmptyComponent={renderEmptyState}
/>
```

- `numColumns={3}`: Creates 3-column grid automatically
- `keyExtractor`: Unique ID for each photo (performance optimization)
- `ListEmptyComponent`: Shows “No photos yet” message

3. Photo Item Press Handling

```
const handlePhotoPress = (photo: Photo) => {
  Haptics.impactAsync(Haptics.ImpactFeedbackStyle.Light);
  if (onViewPhoto) {
    onViewPhoto(photo);  // Opens full-screen viewer
  }
};

const handlePhotoLongPress = (photo: Photo) => {
  Haptics.impactAsync(Haptics.ImpactFeedbackStyle.Medium);
  setSelectedPhoto(selectedPhoto === photo.id ? null : photo.id);
};
```

- **Single Tap**: Opens photo in full-screen viewer
- **Long Press**: Selects photo for delete/share
- **Different Haptics**: Light for tap, medium for long-press

4. PhotoGridItem Component

```
function PhotoGridItem({ photo, isSelected, onPress, onLongPress }) {
  const scale = useSharedValue(1);

  const handlePressIn = () => {
    scale.value = withSpring(0.95, {
      damping: 10,
      stiffness: 300,
    });
  };

  const handlePressOut = () => {
    scale.value = withSpring(1, {
      damping: 12,
      stiffness: 400,
    });
  };
  // ...
}
```

- Each grid item has its own scale animation
- Scales down to 0.95x on press, springs back to 1x on release
- Independent animations for each photo

Key Learning Points

- **Modal Patterns**: Slide-up from bottom is iOS standard
- **FlatList Optimization**: Only renders visible items
- **Nested Components**: PhotoGridItem is defined inside for encapsulation

- **Prop Threading**: onViewPhoto connects gallery to main screen
- **Selection State**: Single source of truth (selectedPhoto ID)

Component 4: `PhotoViewer.tsx`

Purpose Full-screen immersive photo viewer with tap-to-hide controls.

How It Works **1. Modal Presentation**

```
<Modal
  visible={visible}
  transparent
  animationType="none"
  onRequestClose={handleClose}
  statusBarTranslucent
>
```

- `transparent`: Allows custom backdrop
- `animationType="none"`: We handle animation manually
- `statusBarTranslucent`: Full-screen on Android

2. Tap-to-Toggle Controls

```
const [showControls, setShowControls] = useState(true);

const toggleControls = () => {
  setShowControls(!showControls);
};

<Pressable style={StyleSheet.absoluteFill} onPress={toggleControls}>
  <Image ... />
</Pressable>
```

- Tapping image toggles `showControls`
- Controls (top bar, bottom actions) conditionally render
- Creates immersive viewing experience

3. Fade Animation

```
useEffect(() => {
  if (visible) {
    opacity.value = withTiming(1, { duration: 300 });
    scale.value = 1;
    translateX.value = 0;
    translateY.value = 0;
  } else {
    opacity.value = withTiming(0, { duration: 200 });
```

```
  }
}, [visible]);
```

- Fades in when opened
- Resets transform values (for future Pan/Pinch gestures)
- Fades out when closed

4. Action Handlers

```
const handleClose = () => {
  Haptics.impactAsync(Haptics.ImpactFeedbackStyle.Light);
  opacity.value = withTiming(0, { duration: 200 }, () => {
    runOnJS(onClose)();
  });
};
```

- **runOnJS**: Bridges from UI thread to JS thread
- Animates first, then calls `onClose` callback
- Ensures smooth animation before state change

Key Learning Points

- **Modal Nesting**: PhotoViewer modal can be above GalleryModal
- **Conditional Rendering**: if (showControls) pattern
- **runOnJS**: Required to call JS functions from worklets
- **Callbacks**: Animation completion callbacks for sequencing

---

Component 5: `app/index.tsx` (Main Screen)

Purpose Root component that orchestrates camera, UI, and photo management.

State Architecture

```
// Camera state
const [flash, setFlash] = useState<'auto' | 'on' | 'off'>('auto');
const [isTorchOn, setTorchOn] = useState(false);
const [cameraReady, setCameraReady] = useState(false);

// Photo state
const [isCapturing, setIsCapturing] = useState(false);
const [latestPhoto, setLatestPhoto] = useState<Photo | null>(null);
const [allPhotos, setAllPhotos] = useState<Photo[]>([]);

// UI state
const [showGallery, setShowGallery] = useState(false);
const [viewingPhoto, setViewingPhoto] = useState<Photo | null>(null);
```

Flow of Control 1. Component Mounting

```
useEffect(() => {
  loadPhotos();
}, []);

const loadPhotos = async () => {
  const photos = await PhotoManager.getPhotos();
  setAllPhotos(photos);
  const latest = await PhotoManager.getLatestPhoto();
  setLatestPhoto(latest);
};
```

- Runs once on mount (`[]` dependency)
- Loads all photos from storage
- Sets latest photo for thumbnail

2. Photo Capture Flow

```
User taps CaptureButton
  ↓
takePhoto() called
  ↓
setIsCapturing(true) → Shows progress ring
  ↓
camera.current.takePhoto() → Captures image
  ↓
PhotoManager.savePhoto(path) → Saves to storage
  ↓
setLatestPhoto(savedPhoto) → Updates thumbnail
  ↓
loadPhotos() → Refreshes gallery
  ↓
setIsCapturing(false) → Hides progress ring
```

3. Gallery Interaction Flow

```
User taps PhotoThumbnail
  ↓
handleGalleryOpen() → setShowGallery(true)
  ↓
GalleryModal slides up
  ↓
User taps photo in grid
  ↓
handleViewPhoto(photo) → setViewingPhoto(photo)
  ↓
PhotoViewer fades in
  ↓
```

```
User taps close
  ↓
handleViewerClose() → setViewingPhoto(null)
  ↓
PhotoViewer fades out
```

4. Camera Lifecycle Management

```
const isFocused = useIsFocused();
const appState = useAppState();
const isActive = isFocused && appState === "active";

<Camera
  isActive={isActive}
  // ...
/>
```

- **isFocused**: True when this screen is visible
- **appState**: "active" when app is in foreground
- **isActive**: Camera only runs when both are true
- **Why?** Saves battery and prevents background camera access

5. useKeepAwake Integration

```
useKeepAwake();
```

- Prevents screen from sleeping during camera use
- Essential for photo/video capture apps
- Automatically deactivates when component unmounts

Key Learning Points

- **State Lifting**: Common state lives in root, passed down via props
- **Async Operations**: Photo loading and saving are async
- **Conditional Rendering**: Different modals based on state
- **Lifecycle Hooks**: useEffect for mounting, cleanup
- **Derived State**: isActive computed from isFocused and appState

---

Animation System

Animation Library: React Native Reanimated v4

Why Reanimated? Traditional React Native animations run on the **JavaScript thread**, which can drop frames when: - JS thread is busy - Garbage collection occurs - Heavy computations run

Reanimated animations run on the **UI thread**, guaranteeing 60fps.

Core Concepts

1. Shared Values

```
const x = useSharedValue(0);
```

- Live on UI thread
- Can be read/written from both threads
- Don't trigger React re-renders when changed
- Perfect for animations

2. Worklets

```
const animatedStyle = useAnimatedStyle(() => {
  'worklet';  // Optional in v4, auto-detected
  return {
    transform: [{ translateX: x.value }]
  };
});
```

- Functions marked with 'worklet' or inside certain hooks
- Run on UI thread at 60fps
- Can read shared values without overhead
- Cannot access closures from JS thread (use `.value`)

3. Animation Functions **withTiming** - Linear or eased interpolation

```
value.value = withTiming(100, {
  duration: 300,
  easing: Easing.bezier(0.25, 0.1, 0.25, 1),
});
```

withSpring - Physics-based spring animation

```
value.value = withSpring(100, {
  damping: 10,      // Controls bounce (lower = more bounce)
  stiffness: 300,   // Controls speed (higher = faster)
  mass: 1,          // Weight (higher = slower)
});
```

withRepeat - Loop animations

```
value.value = withRepeat(
  withTiming(360, { duration: 1000 }),
  -1,    // -1 = infinite, N = repeat N times
  false  // reverse on each iteration?
);
```

withSequence - Chain animations

```
value.value = withSequence(
  withTiming(100, { duration: 200 }),
  withTiming(50, { duration: 200 }),
  withTiming(0, { duration: 200 })
);
```

4. Entrance Animations

```
<Animated.View entering={FadeInDown.delay(200).duration(600)}>
```

- Built-in entrance/exit animations
- `FadeInDown`, `FadeInUp`, `SlideInLeft`, etc.
- Chainable modifiers (`.delay()`, `.duration()`, `.springify()`)

Animation Patterns Used in WaterMark

Pattern 1: Continuous Rotation

```
// CaptureButton.tsx
rotation.value = withRepeat(
  withTiming(360, { duration: 10000, easing: Easing.linear }),
  -1,
  false
);
```

- Rotates 360° every 10 seconds
- Infinite loop
- Linear easing (constant speed)

Pattern 2: Spring Press/Release

```
// CaptureButton.tsx
const handlePressIn = () => {
  scale.value = withSpring(1.15, { damping: 10, stiffness: 300 });
};

const handlePressOut = () => {
  scale.value = withSpring(1, { damping: 12, stiffness: 400 });
};
```

- Natural, bouncy feel
- Different spring configs for press/release
- Release is slightly stiffer for snappier return

Pattern 3: Sequential Flip

```
// PhotoThumbnail.tsx
rotateY.value = withSequence(
  withTiming(90, { duration: 150 }),  // Flip to edge
```

```
  withTiming(0, { duration: 150 })     // Flip back
);
```

- 2-step animation
- Creates 3D flip effect
- Total 300ms duration

Pattern 4: Modal Slide-Up

```
// GalleryModal.tsx
translateY.value = withSpring(0, {
  damping: 30,
  stiffness: 300,
});
backdropOpacity.value = withTiming(1, { duration: 300 });
```

- Parallel animations (Y position + opacity)
- Spring for modal, timing for backdrop
- Creates polished modal presentation

State Management

State Architecture: Lifted State Pattern

The app uses **lifted state** where common state lives in the root component and flows down via props.

```
index.tsx (Root)
   [photos state]
   [UI state]
   [camera state]

      > PhotoThumbnail (receives: latestPhoto)
      > CaptureButton (receives: isCapturing)
      > GalleryModal (receives: photos, onViewPhoto)
      > PhotoViewer (receives: viewingPhoto)
```

State Categories

1. Camera State

```
const [flash, setFlash] = useState<'auto' | 'on' | 'off'>('auto');
const [isTorchOn, setTorchOn] = useState(false);
const [cameraReady, setCameraReady] = useState(false);
```

- Controls camera hardware
- Updated by user interaction (flash toggle, etc.)
- Used to enable/disable capture button

2. Photo Data State

```
const [latestPhoto, setLatestPhoto] = useState<Photo | null>(null);
const [allPhotos, setAllPhotos] = useState<Photo[]>([]);
```

- Managed by PhotoManager
- Updated after capture, delete, load
- Source of truth for UI

3. UI State

```
const [isCapturing, setIsCapturing] = useState(false);
const [showGallery, setShowGallery] = useState(false);
const [viewingPhoto, setViewingPhoto] = useState<Photo | null>(null);
```

- Controls modal visibility
- Tracks UI loading states
- Determines which overlays show

State Update Patterns

Pattern 1: Derived State

```
const isActive = isFocused && appState === "active";
```

- Computed from other state
- No useState needed
- Recalculates on every render

Pattern 2: Async State Updates

```
const takePhoto = async () => {
  setIsCapturing(true);
  try {
    const photo = await camera.current.takePhoto();
    const savedPhoto = await PhotoManager.savePhoto(photo.path);
    setLatestPhoto(savedPhoto);
    await loadPhotos();
  } finally {
    setIsCapturing(false);
  }
};
```

- Set loading state first
- Perform async operation
- Update data state
- Clear loading state (guaranteed with finally)

Pattern 3: State Synchronization

```
const loadPhotos = async () => {
  const photos = await PhotoManager.getPhotos();
  setAllPhotos(photos);
  const latest = await PhotoManager.getLatestPhoto();
  setLatestPhoto(latest);
};
```

- Single function updates multiple related states
- Keeps UI consistent
- Called after mutations (capture, delete)

Data Flow

Photo Lifecycle

```
User taps
Capture Btn



takePhoto()
- Set isCapturing



camera.takePhoto()
- Returns photo obj



PhotoManager.savePhoto()
- Creates Photo object
- Stores in memory array
- Returns saved Photo



setLatestPhoto()
- Updates thumbnail
```

```
  loadPhotos()
  - Refreshes gallery




  setIsCapturing(false)
```

Component Communication

Parent → Child (Props)

```
// index.tsx passes to GalleryModal
<GalleryModal
  visible={showGallery}
  photos={allPhotos}
  onClose={handleGalleryClose}
  onViewPhoto={handleViewPhoto}
/>
```

- Data flows down
- Callbacks flow up
- Props are the interface

Child → Parent (Callbacks)

```
// GalleryModal calls parent's callback
const handlePhotoPress = (photo: Photo) => {
  if (onViewPhoto) {
    onViewPhoto(photo);  // Calls index.tsx's handleViewPhoto
  }
};
```

- Child doesn't modify parent state directly
- Calls function passed as prop
- Parent handles state update

Sibling Communication

`PhotoThumbnail → (via index.tsx) → GalleryModal`

1. PhotoThumbnail tapped → calls `onPress` prop
2. index.tsx receives callback → `setShowGallery(true)`
3. GalleryModal receives `visible={true}` → opens

Code Patterns & Best Practices

1. TypeScript Interfaces

```
interface CaptureButtonProps {
  onPress: () => void;
  disabled?: boolean;
  isCapturing?: boolean;
}
```

Benefits: - Self-documenting code - Autocomplete in IDE - Compile-time error checking - `?` marks optional props

2. Component Composition

```
function GalleryModal() {
  // Main component logic

  function PhotoGridItem() {
    // Nested component for encapsulation
  }

  return (
    <Modal>
      <FlatList renderItem={renderPhoto} />
    </Modal>
  );
}
```

Benefits: - PhotoGridItem only used here - Access to parent's closures - Keeps related code together

3. Custom Hooks Pattern

```
// Could extract to usePhotoManager.ts
const usePhotoManager = () => {
  const [photos, setPhotos] = useState([]);

  const loadPhotos = async () => {
    const photos = await PhotoManager.getPhotos();
    setPhotos(photos);
  };

  return { photos, loadPhotos };
};
```

Benefits: - Reusable logic - Separates concerns - Cleaner components

4. Error Handling

```
try {
  const photo = await camera.current.takePhoto();
  // ... success path
} catch (error) {
  console.error('Failed to take photo:', error);
} finally {
  setIsCapturing(false);  // Always runs
}
```

Benefits: - Handles failures gracefully - finally ensures cleanup - Prevents stuck loading states

5. Conditional Rendering

```
{hasFlash && (
  <FlashControl />
)}

{showGallery && (
  <GalleryModal />
)}
```

Benefits: - Only render when needed - Saves performance - Clear intent

6. StyleSheet Organization

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#000',
  },
  // ... more styles
});
```

Benefits: - Optimized (styles created once) - Organized (all styles in one place) - Autocomplete for style names

7. useRef vs useState

```
// useState - triggers re-render
const [count, setCount] = useState(0);

// useRef - no re-render
const previousValue = useRef(null);
```

When to use each: - **useState**: UI depends on this value - **useRef**: Just need to store data between renders

8. useEffect Dependencies

```
// Run once on mount
useEffect(() => {
  loadPhotos();
}, []);

// Run when visible changes
useEffect(() => {
  if (visible) {
    animateIn();
  }
}, [visible]);
```

Rule: - Include all values used inside effect - Empty array `[]` = run once - Missing deps = bugs!

Learning Resources

React Native Fundamentals

- React Native Docs
- React Hooks
- TypeScript Handbook

Reanimated

- Reanimated Docs
- Animations Guide
- Spring Physics

Expo

- Expo Docs
- Expo Router
- Expo Modules

Vision Camera

- Vision Camera
- Camera Guides

Key Takeaways

Architecture Lessons

1. **Component-based design** makes code modular and reusable
2. **Lifted state** keeps data flow predictable
3. **Prop drilling** is acceptable for small apps
4. **TypeScript** catches bugs before runtime

Performance Lessons

1. **Reanimated** runs animations on UI thread (60fps)
2. **FlatList** only renders visible items
3. **useCallback/useMemo** prevent unnecessary re-renders (not used yet but important)
4. **Image caching** helps with performance (handled by React Native)

UX Lessons

1. **Haptic feedback** makes interactions feel premium
2. **Spring animations** feel more natural than linear
3. **Loading states** give user feedback
4. **Empty states** guide user when no content

React Native Lessons

1. **Hooks** replace class components
2. **useEffect** manages side effects
3. **StyleSheet** optimizes styling
4. **Platform-specific code** handles iOS/Android differences

---

Next Steps for Learning

Beginner

1. Understand component props and state
2. Learn basic hooks (useState, useEffect)
3. Practice styling with StyleSheet
4. Understand async/await

Intermediate

1. Master Reanimated shared values
2. Build custom hooks
3. Implement gesture handlers
4. Add TypeScript to projects

Advanced

1. Optimize re-renders with useMemo/useCallback
2. Build custom native modules
3. Implement complex animations
4. Add testing (Jest, Detox)

Summary

This WaterMark app demonstrates: - Modern React Native architecture - Advanced animations with Reanimated - TypeScript for type safety - Component composition - State management patterns - Async operations - Native module integration (Camera, Haptics) - Modal presentations - UX best practices

You've learned: - How to structure a real-world React Native app - Animation techniques for premium UX - State management and data flow - Component communication patterns - TypeScript integration - Performance optimization basics

Keep building!