Technical Requirements – Season Status Indicator

Challenge is accessible on Memberspot: https://pl-coding.mymemberspot.io/library/jx3b7Qik9ip5qpNI8IF2/2BiP9k9ZAdvPhLG5wf7P/wzFllGj8Z0LDOUGmibaQ/details

🎭 Scenario

Imagine an app that shows the user the current “state of the world” - for example, the current season. At first, the app doesn’t know anything and stays in a loading state, then it receives a system-defined state, and later allows the user to change it manually. In real projects, screens like this often break: the UI doesn’t update in time or shows the wrong state. In this challenge, your task is to write tests that verify the UI always reflects the current state correctly.

👉 GitHub Repository

The implementation for this mini-challenge is provided in a GitHub repository: https://github.com/PL-Coding-GmbH/Campus-SpringValidation/tree/challenge/season-status-indicator

The repository contains multiple branches. Each branch corresponds to a **separate mini-challenge** in this series.

Clone the repository, switch to the branch for the current challenge, and work with the existing code. Your task is to focus on writing tests - the implementation itself should not be modified.

🎯 Feature Goal

The goal of this challenge is to write UI tests that validate correct UI state rendering and transitions across loading, winter, and spring states. You must ensure that the screen consistently reflects the current state, including toggle behavior, images, status text, and interaction availability. The focus is on UI state synchronization, not implementation details.

⚙️ App Behavior Overview

The app displays the current seasonal state, which is resolved in multiple steps and directly affects the UI.

Initial / Loading

- When the screen is opened, the app enters an initial **loading state**.
- During this state:
  - user interaction with the toggle is disabled;

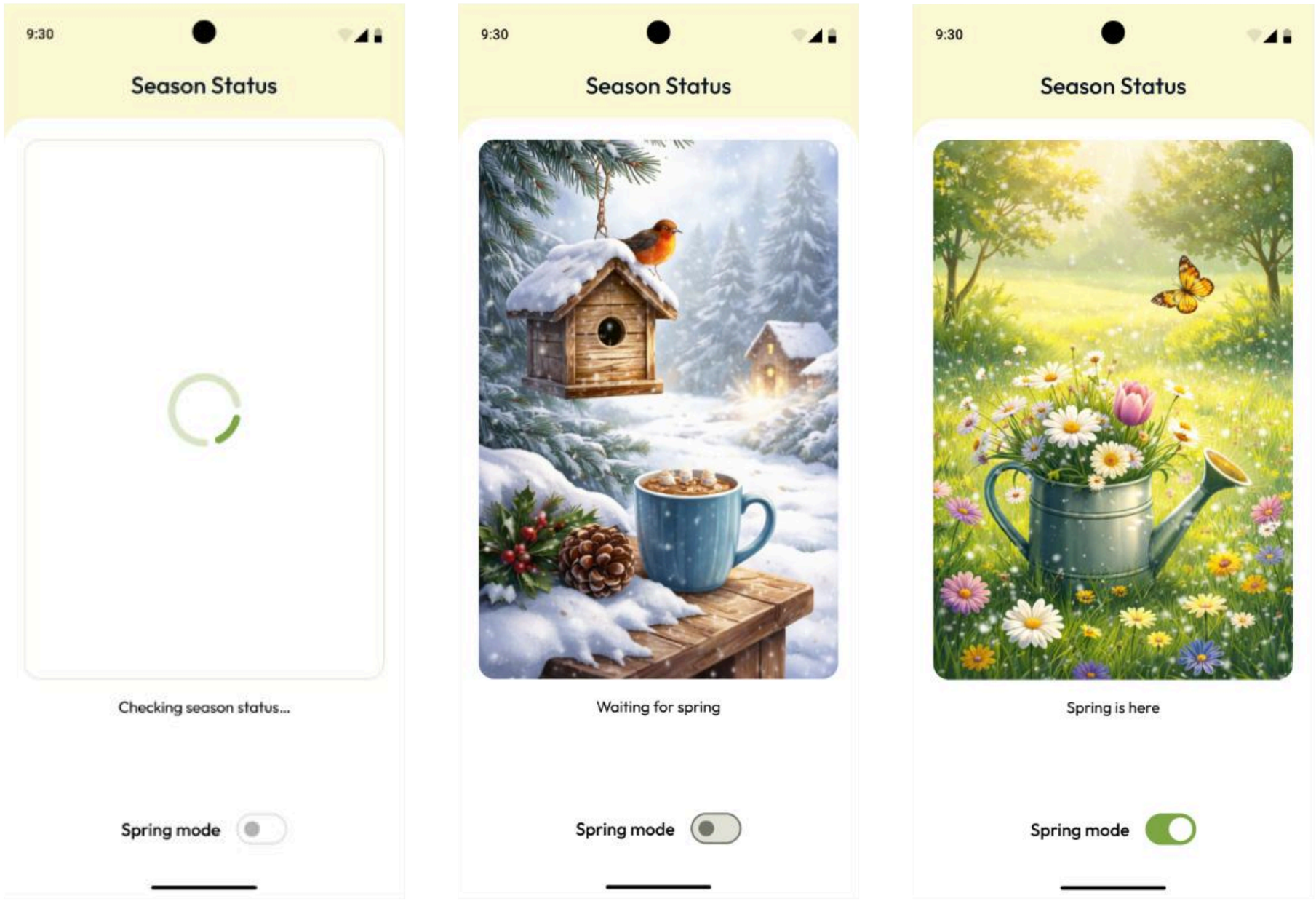
    - a loading indicator is displayed;
- After a short delay (simulating system initialization), the app transitions to the next state.

System-defined state (Winter)

- Once initialization is complete, a **system-defined state** is set — Winter.
- In this state:
    - the toggle becomes enabled;
    - the loading indicator is hidden;
    - the UI displays the winter state and its corresponding status text.
- The user has not manually changed the state yet.

User-driven state (Spring)

- The user can change the state by enabling **Spring mode**.
- When the toggle changes:
    - the state switches between Winter ↔ Spring;
    - the UI should update immediately to reflect the new state.
- After the loading phase, the current state is always determined by the toggle position.



🧪 Validation Tests Requirements

In this mini-challenge, you are required to write UI tests that verify correct screen rendering for each of the main states.

Each test must validate the entire UI state as a whole and **must use the specified test name**.

1️⃣ Initial / Loading State

Test name:

- loadingState_isDisplayedCorrectly

Verify:

- a loading indicator is displayed;
- the toggle is present but **disabled**;
- the status text is exactly **"Checking season status..."**.

2️⃣ Winter State (System-defined)

Test name:

- winterState_isDisplayedCorrectly

Verify:

- the toggle is **enabled** and set to OFF;
- the loading indicator is **not displayed**;
- the winter image is **displayed**;
- the status text is exactly **"Waiting for spring"**.

3️⃣ Spring State (User-driven)

Test name:

- springState_isDisplayedCorrectly

Verify:

- the toggle is **enabled** and set to ON;
- the loading indicator is **not displayed**;
- the spring image is **displayed**;
- the status text is exactly **"Spring is here"**.

ℹ️ Notes

- Test names must be used **exactly as specified**.
- Test execution order does not matter.
- The focus is on UI state correctness, not on individual UI elements.

🤔 What's Allowed?

- Standard Android / Jetpack libraries.
- Any testing approach that verifies validation logic.

⚠️ What's not important

- Writing additional tests beyond those specified in the requirements.
- Covering extra edge cases not mentioned in the challenge.

🔗 Useful Links for This Challenge

- Test your Compose layout
- Compose testing common patterns
- Testing in Jetpack Compose Codelab
- Testing APIs
- The Ultimate Guide to Android Testing

🏆 Submission & Rewards

- Successfully submitting this challenge via the **/submit-challenge** command on Discord grants you **100 XP**.
- Your submission must include:
  a. A **Gist link** with your implementation.
  b. A **screenshot** of the test run results showing:
    - which tests passed,
    - which tests failed,
    - and the names of the executed tests.

> 💡 **Note:**
>
> Some tests in this challenge are **expected to fail** due to intentionally incorrect behavior in the app implementation.
>
> Your goal is to write correct tests, not to make all tests pass.

📩 How to Submit a Mini-Challenge

- In any Discord channel, type **/submit-challenge**.
- Attach your screen recording demonstrating the implementation according to the challenge requirements.
- Supported formats: MP4, MOV, AVI, MKV, WEBM, PNG, JPEG, JPG, GIF.
- The total file size must **not exceed** 50 MB.
- If additional materials are required (e.g. screenshots), attach up to 4 additional image files in the command pop-up before submitting.
- Press **Enter** to send the files.
- In the bot flow, select **Mini-Challenge**.
- Choose the month this mini-challenge belongs to (each month includes five mini-challenges).
- Select the exact **challenge name** you are submitting.
- Submit challenge.