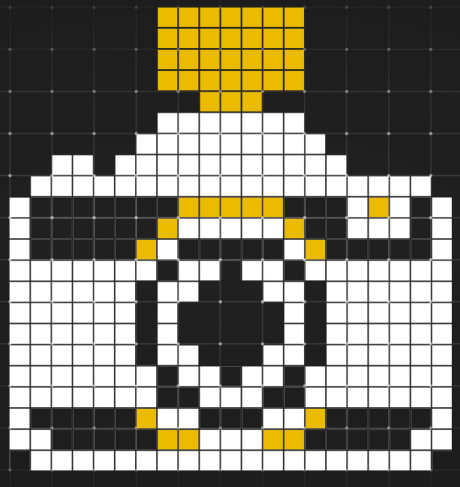
MASTERING ANDROID SCREENSHOT TESTING

RELEASE WITH CONFIDENCE
ELIMINATE VISUAL REGRESSIONS
ACROSS DEVICES AND CONFIGURATIONS



ALEX ZHUKOVICH

Table of Contents

Introduction

Testing mobile applications is a challenge for Android developers, as we need to ensure not only that our application functions as expected but also that it looks exactly as intended across multiple device form factors (phones, foldable devices, tablets) from different vendors and configurations. All components on the screen can behave as expected, but issues can arise, such as wrong alignment or barely readable text caused by insufficient text color contrast in dark mode.

This book introduces the concept of screenshot testing (also known as visual testing, snapshot testing, or pixel-perfect testing), which is an automated testing approach that verifies the look of the components and screens. The process starts by recording a reference image (also known as a "golden" image) that represents the expected design. Then, each time the tests run, they capture the current state of the screen and compare it against that reference. When the user interface (UI) changes, either intentionally or unintentionally, the tests detect these visual changes.

Screenshot testing is a great addition to functional UI tests, which verify the behavior of the application but aren’t able to verify appearance.

This book will focus on fundamental principles that apply regardless of your framework and tools. We will explore the common challenges in screenshot testing and answer the following questions:

- When to use screenshot tests and when not to use screenshot tests?
- What to verify with screenshot tests? What frameworks and tools are available for screenshot testing, and what impact do they have on the development workflow?
- How to generate screenshot tests for `@Preview` functions in Jetpack Compose?
- How to integrate screenshot tests into the continuous integration and continuous delivery (CI/CD) pipeline?
- How to optimize a screenshot testing workflow?

By the end of the book, you will be able to choose a framework that best suits your project. Using an example application, you'll also identify the key components and screens to cover with screenshot tests. This will create a safety net to catch visual bugs before release.

Who This Book Is For

This book is intended for Android developers and mobile QA engineers. Specifically, it targets those who are responsible for ensuring the visual consistency and quality of mobile applications. The book will guide you through implementing automated screenshot testing for Android projects, with the goal of improving application quality and replacing manual visual verification with automated tests.

You'll find this book particularly valuable if you are one of the following:

- An Android developer looking to enhance and improve your testing practices
- A mobile QA engineer focused on Android test automation
- A mobile team lead who wants to improve quality processes
- A developer who is working on a design system
- A QA professional transitioning from manual to automated testing for mobile apps

The book focuses on the challenges faced by Android teams, such as device fragmentation, operating system (OS) version variations, and UI customizations. Previous experience with screenshot testing is not necessary, as we will cover everything from basic to advanced techniques. The content is designed to be helpful for beginners and also beneficial for those with some experience in screenshot testing.

How to Read This Book

This book focuses on core concepts and best practices, with additional resources available online. As this book is published incrementally, some chapters are still in progress and marked as "coming soon." You'll receive updates as new content is

added.

The book is organized into five parts:

Part 1: Introduction to Screenshot Testing

Part 1 introduces you to the mobile testing world. Here you will explore the challenges in mobile testing, different types of tests, and the role and importance of screenshot testing.

Part 2: Preparation for Screenshot Testing Automation

Part 2 explains how to establish standards for your team or your organization and choose the right tools and frameworks for screenshot testing for your Android projects. We will also deep dive into all processes related to screenshot testing and explore different ways of setting up the emulators and real devices.

Part 3: Implementing Screenshot Testing (some chapters coming soon)

Part 3 explains in detail, with examples, how to test components and screens in Android projects using various frameworks and tools. After reading this part, you will know how to verify your components and screens with different font sizes, themes, and locales. Besides that, you will learn how to create test cases for different devices, verify animations, and address common screenshot testing challenges.

Part 4: Advanced Screenshot Testing Techniques (coming soon)

Part 4 explains advanced screenshot testing techniques, like parameterized testing and different strategies for storing screenshots. Additionally, it covers how to integrate screenshot testing into your CI/CD workflow. We will also explore how to optimize the screenshot testing workflow, reduce execution time, and introduce the possibility of executing tests for a specific feature only.

Part 5: Future Trends and Best Practices (coming soon)

Part 5 explores the future trends in screenshot testing, AI tools, and recommendations for introducing screenshot testing to Compose Multiplatform projects.

Appendix A: Reference Guide for Local and Instrumentation Tests

A quick reference on the differences between local and instrumentation tests on Android.

Appendix B: Reference Guide for Non-functional Testing Types

A quick reference with definitions of common non-functional testing types.

The best way to learn about screenshot testing is to create test cases for an application. We will analyze a mood tracker application that was built for demonstration in this book and add screenshot tests for this application. This app allows users to track their mood and see insights based on their data.

What You Need to Use This Book

To use the code samples in this book, you will need to download the latest stable release of Android Studio. You can also use other integrated development environments (IDEs) or build the app/run test cases from the command line. However, I assume that you'll use Android Studio.

You do not need a physical Android device to run tests because you can run them locally (on your machine) or on an emulator.

Part 1: Introduction to Screenshot Testing

There are many similar challenges involved with testing software products across platforms, but each platform also has its own specific cases. Android apps work on devices from many vendors, and manufacturers often introduce their own optimizations and limitations. Companies like Google, Samsung, and Huawei offer proprietary services, and users may want to choose a vendor depending on the ecosystem around their device.

On top of that, how developers build the UI also varies. Jetpack Compose is becoming the standard for native Android development; it helps many developers create UI components and screens using a declarative way of building a user interface. Jetpack Compose was built from scratch, and it doesn't use the View framework (a solution for building a UI that was available from the first release of Android). It has an impact on testing applications because you can easily find an application that uses "traditional" (application is fully built with a View framework), "modern" (app is built entirely using Jetpack Compose), and "combined" (application uses the View framework and Jetpack Compose) approaches to creating UIs.

This part introduces you to the world of mobile testing and covers different types of tests and important concepts related to Android testing. You will also focus on understanding the role of screenshot tests for improving the quality of applications; this will help you improve trust in your brand.

This part includes the following chapters:

- Chapter 1: Mobile Testing Fundamentals
- Chapter 2: Introduction to Screenshot Testing

Chapter 1: Mobile Testing Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Chapter 2: Introduction to Screenshot Testing

The visual quality of any application is an important aspect of the first impression and enjoyment of using it. If users like the look and feel of the app, they tend to explore the application more and, in the end, continue instead of switching to alternatives. Visual quality is one of the important aspects for any company that wants to build a solid brand around their product.

Imagine a customer who needs to create an account in your application, but the text on the button is not readable, as illustrated in Figure 2-1. There is a high chance that some users will try to find alternatives to this product. In this case, the source of doubt is a visual quality that might cause the user to never explore features of the application.

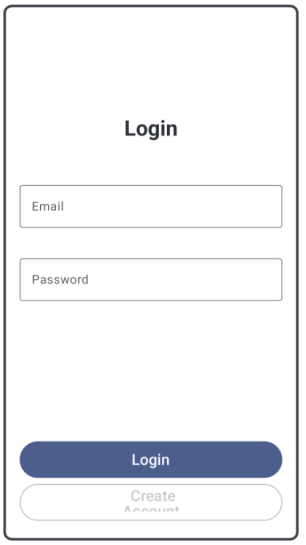


Figure 2-1. Login screen: Issue with the Create account button

Usually, companies spend a lot of resources on customer acquisition, and visual issues in a software product can easily ruin all these efforts.

When the project is at an early stage, it's possible to verify every screen of the application manually in a short amount of time. But when you invest a lot of resources in supporting multiple device form factors, light and dark modes, and various font sizes, you cannot quickly verify all possible combinations for every screen. In this chapter, we will discuss screenshot testing that can help you verify

all these combinations quickly and be sure that your application looks as expected.

Understanding Screenshot Testing

The main goal of screenshot testing is to confirm that the software product has the expected appearance, including positioning of components, correct colors and component sizes, correct layouts for different form factors, and no localization issues for different languages and locales. These cases can be verified using both manual and automated testing approaches. However, you should remember that validating an application on a single device, without changing the color mode and font size, is usually not enough to be sure that the application doesn't have any visual imperfections.

There are multiple UI test types that interact with the user interface, but do not verify the visual accuracy of the application:

End-to-end tests

End-to-end tests ensure that the application works in the production environment. This means that all application layers work correctly.

UI tests with fake data

UI tests with fake data verify that all components appear for different states of the screen (no data, successfully loaded data, network is unavailable, and much more).

Accessibility tests

Accessibility tests check whether everyone can use a user interface, regardless of their abilities.

End-to-end tests, UI tests with fake data, and accessibility tests can pass when the UI has visual problems.

If you use Jetpack Compose, you likely already use `@Preview` to check how components or screens look during development. The

> key limitation is that previews require manual verification and don't automatically catch regressions. Screenshot tests complement previews by introducing automated visual verification into your test pipeline.

As shown in Figure 2-2, the human eye can easily find issues that automation tests miss, even when both screens display similar data.

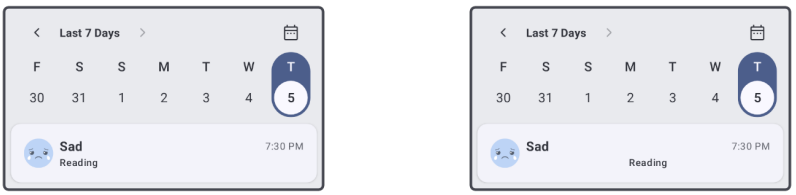


Figure 2-2. Demo: Correct vs. incorrect component rendering

To verify visual quality and ensure all components are positioned correctly, we need to use screenshot testing (also commonly called visual testing, snapshot testing, or pixel-perfect testing). In Figure 2-2, we can see that "Reading" is left-aligned in the correct rendering, but centered in the incorrect one; for our app, we want the left alignment.

There are multiple approaches to how screenshot testing can be done, but not all of them are available for every platform because of the implementation of the UI layers on different platforms and the available frameworks for these platforms.

Screenshot testing compares the expected appearance of a component or screen with its current state. In Chapter 4, we will explore frameworks available for visually testing Android projects.

Some testing tools allow comparing Cascading Style Sheets (CSS) properties of elements, such as color, font size, and border radius. This approach is mainly used in web development, where the styles of elements can be easily inspected and verified. However, as components become more complex, visualizing the UI mentally from a list of individual property assertions is becoming harder—unlike screenshot tests, which provide an immediately understandable visual comparison. This method isn't available in Android development.

AI tools also can be used for evaluating changes in rendering the component of the screen. We will talk about this topic in Chapter 19 (coming soon).

In many projects, everything starts with a manual comparison of components and screens against the design. When the application is growing and supports different color schemes and device form factors, and is adapted for various font sizes, this can become a time-consuming operation, and screenshot testing starts to be more efficient.

Use Cases

There are multiple use cases for adding automated screenshot testing to a project on a temporary or permanent basis. The important factors to consider while introducing screenshot testing to the project are related to cost (development, maintaining screenshot tests, and test execution).

Screenshot tests are particularly valuable during the refactoring of the UI layer, where they serve as a safety net to detect unintended visual regressions when restructuring component internals, optimizing rendering logic, or reorganizing screen layouts. By even temporarily implementing these tests during major changes, developers can confidently refactor underlying code while ensuring the user interface maintains its intended appearance and behavior throughout the refactoring process.

There are also many product requirements that can be covered by screenshot testing, and you can use it to be confident that your application looks and feels as expected:

- Supporting multiple device form factors
- Supporting light and dark modes
- Supporting multiple color schemes
- The application content adapts to different font sizes
- The application supports advanced levels of customization and personalization

- The application supports localization and internationalization
- The application uses A/B testing with different UI variants

A/B testing complicates manual verification. While both versions need to be visually accurate, switching between them to check is time-consuming. Typically, a developer or QA engineer must manually change configuration flags to verify one variant at a time. Screenshot tests can cover both variants independently, ensuring each one renders correctly without manual configuration changes.

Manually verifying various combinations of these requirements takes a lot of time. Let's take a look at a simplified example of testing an application screen, which has three states (*empty* (no data available), *loading*, and *success* (data is available)) and supports dark and light modes and two languages (Figure 2-3).

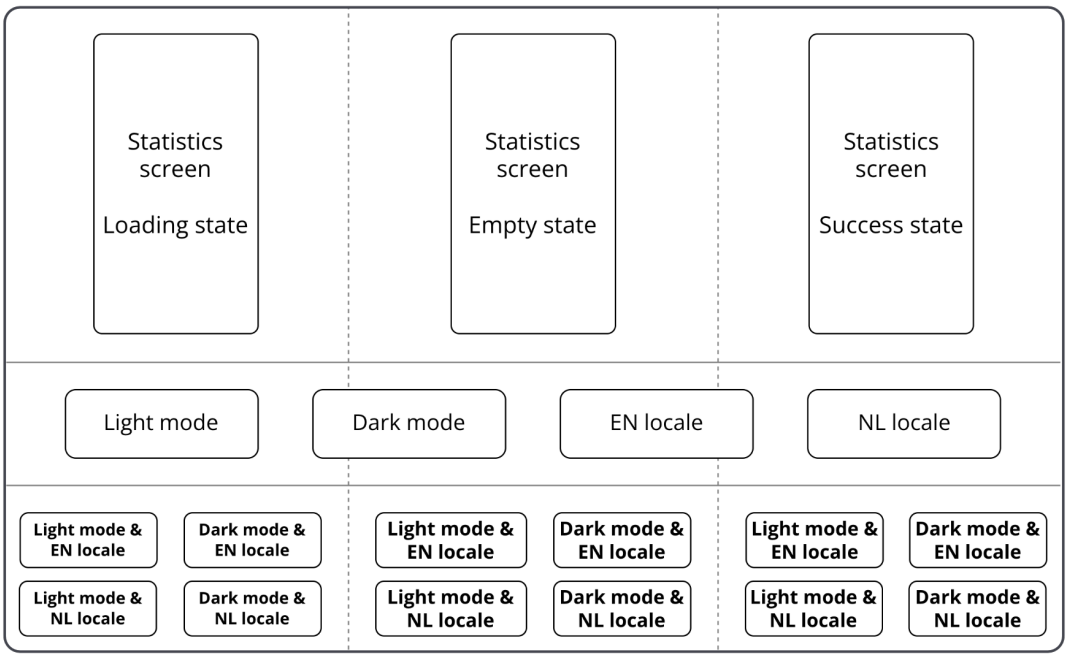


Figure 2-3. Test case for visual verification of the application screen

As you can see, even here, we need to verify 12 variations of the Statistics screen. We excluded verification screens on various device form factors and checking of

adaptability for different font sizes. For many teams, the manual work doesn't stop here. Developers frequently attach screenshots of an implemented feature to their pull requests (see Figure 2-4).

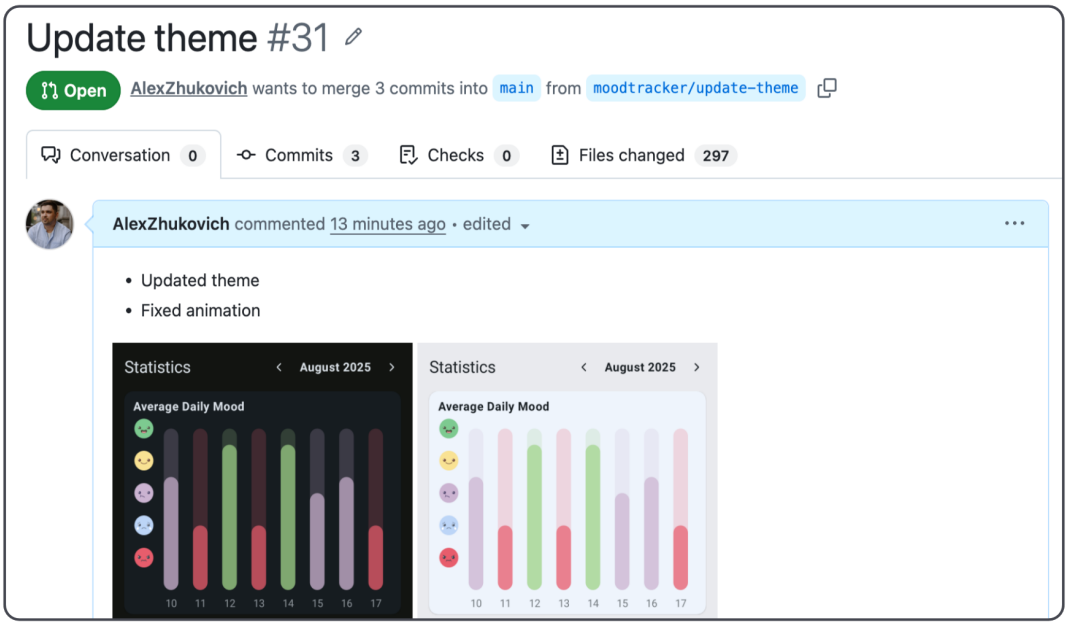


Figure 2-4. Pull request with manually added screenshots

In practice, these screenshots usually show a single locale and a configuration—one device, one color mode, and one locale. Once screenshot tests are implemented, the process changes: screenshots are automatically generated and added, or a link to the tool that stores screenshots is included, ensuring coverage for all configurations.

Screenshot tests have their own costs, as do any other types of tests, but covering the crucial features with such tests will guarantee that you don't have visual imperfection and your brand won't be associated with a lack of visual quality.

There are a few areas of mobile applications that can be verified using screenshot testing:

Design system and custom components

A design system is a set of components, icons, and theme attributes (shapes, colors, and typography). The simplest version of the design system is a set of cus-

tom components that are reused in multiple features of the application. Even if a project has a design system, not every component is included, as certain components are specific to a single screen.

Screens of applications

Screens of applications are an essential part of every application. Even if components are tested, this doesn't guarantee that screens do not have any imperfections in their positioning on a screen.

Widgets

Widgets are an optional element in many applications. However, they're perfect candidates for screenshot testing, as users will notice any misalignment or inconsistencies in their placement.

Custom content in notifications

Custom content in notifications is optional, and not used very often, but if we use them, we want to be sure that our notifications look polished.

This comprehensive testing approach becomes even more critical when considering the market for mobile devices, which is incredibly versatile. Devices from different vendors use different versions of operating systems, creating a complex testing landscape that requires thorough visual verification. In addition to that, there are multiple form factors for mobile devices to consider, including phones, tablets, foldable/flippable devices, etc.; Android apps also can work on Windows and Chrome OS devices. So, as you can see, there are multiple variables to consider when we think about the devices and environment where our application can be executed.

To explore usage statistics for mobile devices, you can use the gs.statcounter.com website. The device vendor market share, OS market share, and form factor market shares (as of May 2026) are depicted in Tables 2-1 and 2-2.

Table 2-1. Market shares for the most popular vendors producing mobile devices.

Vendor	Market share
Apple	32.55%
Samsung	19.11%
Xiaomi	8.69%
OPPO	5.59%
Vivo	5.55%
Realme	3.76%

Table 2-2. Market shares for the most popular form factors.

Device type	Market share
Mobile	52.8%
Desktop	45.61%
Tablet	1.59%

In addition to global data, the application's analytics is a great source of information, which can be used for selecting the best devices to focus on when checking the quality of your application, as you want to be sure that your product works perfectly on the most-used devices.

Screenshot testing allows you to verify the visual quality of crucial screens and UI components, which are reused across the application under different conditions, like various color modes, themes, font sizes, device form factors, locales, and more. When you have automated screenshot tests, you will be notified about any visual regressions in your application.

When Not to Use Screenshot Testing

The main limitation of screenshot testing is that the content must be identical between runs, which requires you to use static data for everything displayed on the screen. Let's take a look at Figure 2-5, which shows the home screen of the mood tracker application.

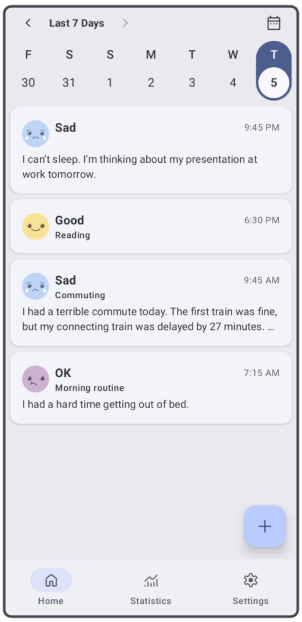


Figure 2-5. Home screen of the mood tracker application

To cover such a screen with data using screenshot tests, you need to use fake data. If you want to recreate the same screen, you need to not only pass the same mood entries and bottom navigation items, but also use the same selected dates and device parameters (light theme, default font size). Of course, the device or emulator should be the same, as screenshot resolution can be different for every device.

So, if a project doesn't allow you to inject fake data, the screenshot tests will be very unstable and are not recommended if the selected framework doesn't support region exclusion. However, in this case, you can add screenshot tests to a design system and custom components to ensure pixel-perfect rendering on the components layer.

Another limitation is that not all frameworks allow you to interact with the UI before taking a screenshot, as different frameworks are implemented internally differently. However, here, you should choose a particular framework if it is important for you. We will talk about various frameworks in Chapter 4.

Another common misconception has to do with the combination of functional UI tests with screenshot tests. For instance, you log in to the application, do some interaction, and at the end, you take a screenshot. This approach is not recom-

mended because it makes your tests more fragile, as there can be multiple points of failure in the functional UI test part and during the screenshot comparison. Maintaining such test cases can be a tricky task, because UI tests and screenshot tests have different limitations, and what is possible for UI tests can be impossible in screenshot tests. For instance, you can use randomly generated data for UI tests, but if the data appears visually, the dynamic content will cause your screenshot tests to fail. On top of that, background threads and animations can introduce timing issues, meaning a screenshot might capture a loading state, a half-finished transition, or a partially rendered screen.

Impact on the Development Process

When you add an additional step to application verification, it affects the development process. It's important to know the pros and cons of adding any dependency to the project.

Screenshot testing has its own cost, and you need to know both sides of the coin before adding it to the project. As we explored before, test cases can be added temporarily or permanently to the project.

Let's compare the pros and cons of adding screenshot tests to the project.

Positive Impacts of Screenshot Tests

Screenshot tests ensure a consistent appearance of application components and screens across different devices and configurations. Screenshot testing of components prevents unnoticed changes before release. They can also serve as a safety net during UI refactoring—by running tests locally before and after the changes, you can quickly verify that no visual regressions were introduced.

Screenshot tests enhance test efficiency. When we speak about UI verification in modern applications, often the behavior of the application is verified using end-to-end (E2E) and UI tests with fake data. They help you to verify the behavior of the application and different states of the screen, but they cannot verify the correctness of rendering or positioning of elements on the screen. The screenshot tests can help solve this problem and guarantee brand consistency of the product.

Screenshot tests improve collaboration across developers, designers, and QA teams. When screenshot tests are included in the development process and a feature is implemented, screenshots are an essential part of every pull request (PR). It means that developers can see the UI of implemented features, designers can verify the UI of the application without installing every version of application on their devices, and the QA team can verify how the testing was implemented and provide feedback related to testing a feature (if it was done by developers).

Screenshot tests integrated with CI/CD provide feedback after any changes in the application. Modern software products use many dependencies, and updating one of them can affect the rendering of your component or screen. If one of the components started rendering differently after updating any dependency, you will be notified about it, and you need to decide whether to update screenshots in your tests or customize components according to your requirements. It helps you to avoid unnoticed changes in the UI of your product.

Negative Impacts of Screenshot Tests

Screenshot tests require manual verification and maintenance. If there are frequent UI changes, screenshot tests may require a lot of maintenance if you need to regenerate them manually. Modern frameworks for screenshot testing allow you to regenerate screenshots for all test cases by running a single command. Newly generated or updated screenshots need manual verification, which takes extra time. However, generated screenshots can be shared with designers to perform a design review, a step that is usually part of their workflow anyway. On the other hand, even if the UI of your application changes often, the UI components in the design system rarely change as frequently.

Screenshot tests can be flaky. Such tests can be flaky because of various factors, like the screen containing dynamic content, rendering issues, or test cases that were executed on devices/emulators with different screen resolutions or device orientations. Both component-level and full-screen screenshots are valuable, but capturing more than what you're verifying increases the chance of instability. In addition to that, selecting the right tool is an important step in reducing these risks. We will talk more about this in Chapter 4, together with discussing

approaches to avoid flaky tests when your team members use different OSs (Windows, MacOS, Linux).

Screenshot tests shouldn't contain screens with dynamic content. When you create a screen test, you should avoid any dynamic content because, at the end, you compare two screenshots. Examples of dynamic content include randomly generated content, user content from remote data sources, advertisements, and the current date or time. It's important to have the option to replace dynamic content with static content for screenshot tests.

Not all frameworks allow you to set a device configuration in tests. Various frameworks use different approaches for doing screenshot testing of your application. Some of them require a real device or emulator; another type uses the same library used for rendering previews inside Android Studio; and the third approach executes tests on the JVM. As a result, every approach has its own limitations, and if the selected framework requires a device or emulator, it means that the continuous integration (CI) environment and every team member who executes test cases should have a similar device, because if the screen resolution is different, the screenshots won't match. We will compare the features, limitations, and approaches to comparing screenshots for the most popular frameworks in Chapter 4.

Screenshot tests can be resource-intensive in a CI environment. If you use a framework that requires a device or emulator, your screenshot tests can require a lot of resources on CI, and if you execute all test cases at once, it can require a lot of time.

As you can see, screenshot tests have pros and cons. Balancing the positive and negative impacts requires planning, selecting the right tools for your case, and maintenance to ensure that screenshot testing improves the development process for your project. In Chapter 6, we will explore what to verify with screenshot tests.

Key Takeaways

- Visual quality significantly affects user first impressions and retention, as

users who encounter unreadable buttons or poor visual experiences often try to find an alternative application.

- Screenshot testing confirms that software products have the expected appearance, including component positioning, correct colors and sizes, and proper layouts across form factors.
- Manual verification becomes impractical when supporting multiple device form factors, light/dark modes, various font sizes, and localization options, as even simple screens require testing multiple combinations of various parameters.
- Screenshot testing applies to design systems (reusable components and themes), application screens (component positioning verification), widgets (optional but crucial elements), and custom notification content to ensure a polished appearance.
- Screenshot testing requires identical screenshot content and consistency across the devices on which test cases are executed.
- Screenshot tests can be a safety net for UI refactoring. Even adding them temporarily on your development machine during major changes can help catch unintended visual regressions.

Part 2: Preparation for Screenshot Testing Automation

After understanding different types of tests, and how screenshot tests can improve the quality of your application, we will explore how to choose frameworks and tools that will simplify the testing process and will suit the project.

In this part, you will also learn how to establish standards and guidelines for the team, department, or organization for the screenshot testing processes, and how to set up devices or emulators if you decide to use instrumentation tests.

By reading this part, you will gain a high-level understanding of the screenshot testing process and learn what to test in the mood tracker application. This knowledge will help you apply the same approach to your own projects. This part includes the following chapters:

- Chapter 3: Establishing Standards and Guidelines for Screenshot Testing
- Chapter 4: Selecting Frameworks and Tools for Screenshot Testing
- Chapter 5: Choosing Between Emulators and Physical Devices for Testing
- Chapter 6: Defining the Scope of Screenshot Testing

Chapter 3: Establishing Standards and Guidelines for Screenshot Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Chapter 4: Selecting Frameworks and Tools for Screenshot Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Chapter 5: Choosing Between Emulators and Physical Devices for Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Chapter 6: Defining the Scope of Screenshot Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Part 3: Implementing Screenshot Testing

By now, you should be familiar with the role of screenshot tests and better equipped to select a framework.

In Part 3, you'll learn how to implement test cases that will verify UI components and screens of an Android application. You will learn how to generate screenshot tests for your preview functions, verify light and dark modes, and the adaptability of screens to system font sizes and locales. Additionally, you will explore how to solve common challenges while adding screenshot tests to your project.

After reading this part, you will know what to test in Android applications using screenshot tests, and you will know how to verify different visual aspects of your Android application to ensure that your app doesn't have any visual imperfections.

This part includes the following chapters:

- Chapter 7: Writing and Executing First Screenshot Tests
- Chapter 8: Handling Asynchronous Images
- Chapter 9: Generating Screenshot Tests for Preview Functions
- Chapter 10: Testing Screens in the Application
- Chapter 11: Testing Across Configurations (coming soon)
- Chapter 12: Testing Across Different Devices (coming soon)
- Chapter 13: Testing Animations (coming soon)
- Chapter 14: Addressing Common Challenges (coming soon)

Chapter 7: Writing and Executing First Screenshot Tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Chapter 8: Handling Asynchronous Images

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Chapter 9: Generating Screenshot Tests for Preview Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Chapter 10: Testing Screens in the Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Appendix A: Reference Guide for Local and Instrumentation Tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.

Appendix B: Reference Guide for Non-functional Testing Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/mastering-android-screenshot-testing.