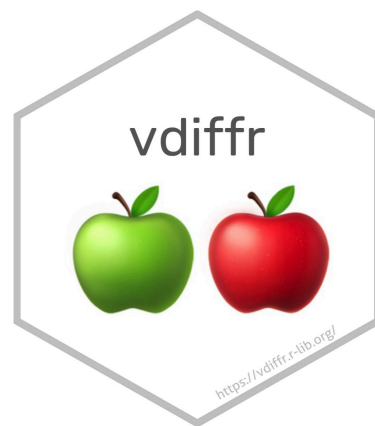# Introduction to Snapshot Testing in R

Indrajeet Patil

Unit testing

The goal of a unit test is to capture the *expected* output of a function using *code* and making sure that *actual* output after any changes matches the expected output.

`{testthat}` is a popular framework for writing unit tests in R.

⚠ **Benefits of unit testing**

- insures against unintentionally changing function behaviour
- prevents re-introducing already fixed bugs
- acts as the most basic form of developer-focused documentation
- catches breaking changes coming from upstream dependencies
- etc.

💡 **Test output**

Test pass only when actual function behaviour matches expected.

actual	expected	tests
📄	📄	✓
📄	📄	✗

Unit testing with `{testthat}`: A recap

⚠ Test organization

Testing infrastructure for R package has the following hierarchy:

Component	Role
📄 **Test file**	Tests for `R/foo.R` will typically be in `tests/testthat/test-foo.R`.
⚗ **Tests**	A single file can contain multiple tests.
≡ **Expectations**	A single test can have multiple expectations.

💡 Example test file

- Every test is a call to `testthat::test_that()` function.

- Every expectation is represented by `testthat::expect_*()` function.

- You can generate a test file using `usethis::use_test()` function.

```
 1  # File: tests/testthat/test-op.R
 2
 3  # test-1
 4  test_that("multiplication works", {
 5    expect_equal(2 * 2, 4) # expectation-1
 6    expect_equal(-2 * 2, -4) # expectation-2
 7  })
 8
 9  # test-2
10  test_that("addition works", {
11    expect_equal(2 + 2, 4) # expectation-1
12    expect_equal(-2 + 2, 0) # expectation-2
13  })
14
15  ...
```

What is different about snapshot testing?

A **unit test** records the code to describe expected output.

(actual) ↔ (expected)

A **snapshot test** records expected output in a separate, human-readable file.

(actual) ↔ (expected)

Why do you need snapshot testing?

If you develop R packages and have struggled to

- test that text output *prints* as expected

- test that an entire file *is* as expected

- test that generated graphical output *looks* as expected

- update such tests *en masse*

then you should be excited to know more about *snapshot tests* (aka *golden tests*)! 🤩

Prerequisites

Familiarity with writing unit tests using `{testthat}`.

If not, have a look at this chapter from *R Packages* book.

Testing text outputs

Snapshot tests can be used to test that text output *prints* as expected.

Important for testing functions that pretty-print R objects to the console, create elegant and informative exceptions, etc.

Example function

Let's say you want to write a unit test for the following function:

Source code

```
1  print_movies <- function(keys, values) {
2    paste0(
3      "Movie: \n",
4      paste0("  ", keys, ": ", values, collapse = "\n")
5    )
6  }
```

Output

```
1  cat(print_movies(
2    c("Title", "Director"),
3    c("Salaam Bombay!", "Mira Nair")
4  ))
```

```
Movie:
  Title: Salaam Bombay!
  Director: Mira Nair
```

Note that you want to test that the printed output *looks* as expected.

Therefore, you need to check for all the little bells and whistles in the printed output.

Example test

Even testing this simple function is a bit painful because you need to keep track of
every escape character, every space, etc.

```
1  test_that("`print_movies()` prints as expected", {
2    expect_equal(
3      print_movies(
4        c("Title", "Director"),
5        c("Salaam Bombay!", "Mira Nair")
6      ),
7      "Movie: \n  Title: Salaam Bombay!\n  Director: Mira Nair"
8    )
9  })
```
```
Test passed 🎊
```

With a more complex code, it'd be impossible for a human to reason about what the
output is supposed to look like.

> ⊘ **Important**
>
> If this is a utility function used by many other functions, changing its behaviour would entail *manually* changing expected
> outputs for many tests.
>
> This is not maintainable! 😩

Alternative: Snapshot test

Instead, you can use `expect_snapshot()`, which, when run for the first time, generates a Markdown file with expected/reference output.

```
1  test_that("`print_movies()` prints as expected", {
2    expect_snapshot(cat(print_movies(
3      c("Title", "Director"),
4      c("Salaam Bombay!", "Mira Nair")
5    )))
6  })
```

```
── Warning: `print_movies()` prints as expected ─────────────────────────
Adding new snapshot:
Code
  cat(print_movies(c("Title", "Director"), c("Salaam Bombay!", "Mira Nair")))
Output
  Movie:
    Title: Salaam Bombay!
    Director: Mira Nair
```

> ⚠️ **Warning**
>
> The first time a snapshot is created, it becomes *the truth* against which future function behaviour will be compared.
>
> Thus, it is **crucial** that you carefully check that the output is indeed as expected. 🔍

Human-readable Markdown file

Compared to your unit test code representing the expected output

```
1   "Movie: \n   Title: Salaam Bombay!\n   Director: Mira Nair"
```

notice how much more human-friendly the Markdown output is!

```
1   Code
2     cat(print_movies(c("Title", "Director"), c("Salaam Bombay!", "Mira Nair")))
3   Output
4     Movie:
5       Title: Salaam Bombay!
6       Director: Mira Nair
```

It is easy to *see* what the printed text output is *supposed* to look like. In other words, snapshot tests are useful when the *intent* of the code can only be verified by a human.

> (i) **More about snapshot Markdown files**
>
> - If test file is called `test-foo.R`, the snapshot will be saved to `test/testthat/_snaps/foo.md`.
>
> - If there are multiple snapshot tests in a single file, corresponding snapshots will also share the same `.md` file.
>
> - By default, `expect_snapshot()` will capture the code, the object values, and any side-effects.

What test success looks like

If you run the test again, it'll succeed:

```
1  test_that("`print_movies()` prints as expected", {
2    expect_snapshot(cat(print_movies(
3      c("Title", "Director"),
4      c("Salaam Bombay!", "Mira Nair")
5    )))
6  })
```

```
Test passed 🎊
```

> ⓘ **Why does my test fail on a re-run?**
>
> If testing a snapshot you just generated fails on re-running the test, this is most likely because your test is not deterministic. For example, if your function deals with random number generation.
>
> In such cases, setting a seed (e.g. `set.seed(42)`) should help.

What test failure looks like

When function changes, snapshot doesn't match the reference, and the test fails:

Changes to function

```
1  print_movies <- function(keys, val
2    paste0(
3      "Movie: \n",
4      paste0(
5        "  ", keys, "- ", values,
6        collapse = "\n"
7      )
8    )
9  }
```

Failure message provides expected (−) vs observed (+) diff.

Test failure

```
1  test_that("`print_movies()` prints as expected", {
2    expect_snapshot(cat(print_movies(
3      c("Title", "Director"),
4      c("Salaam Bombay!", "Mira Nair")
5    )))
6  })
```

```
── Failure: `print_movies()` prints as expected
─────────────────────────────────────
Snapshot of code has changed:
old[2:6] vs new[2:6]
    cat(print_movies(c("Title", "Director"), c("Salaam Bombay!", "Mira
Nair")))
  Output
    Movie:
-     Title: Salaam Bombay!
+     Title- Salaam Bombay!
-     Director: Mira Nair
+     Director- Mira Nair

* Run `testthat::snapshot_accept('slides.qmd')` to accept the change.
* Run `testthat::snapshot_review('slides.qmd')` to interactively review
the change.

Error:
! Test failed
```

Fixing tests

Message accompanying failed tests make it explicit how to fix them.

- If the change was *deliberate*, you can accept the new snapshot as the current *truth*.

```
1  * Run `snapshot_accept('foo.md')` to accept the change
```

- If this was *unexpected*, you can review the changes, and decide whether to change the snapshot or to correct the function behaviour instead.

```
1  * Run `snapshot_review('foo.md')` to interactively review the change
```

> 💡 **Fixing multiple snapshot tests**
>
> If this is a utility function used by many other functions, changing its behaviour would lead to failure of many tests.
>
> You can update *all* new snapshots with `snapshot_accept()`. And, of course, check the diffs to make sure that the changes are expected.

Capturing messages and warnings

So far you have tested text output printed to the console, but you can also use snapshots to capture messages, warnings, and errors.

message

```
1  f <- function() message("Some info for you.")
2
3  test_that("f() messages", {
4    expect_snapshot(f())
5  })
```

── Warning: f() messages
───

```
Adding new snapshot:
Code
  f()
Message
  Some info for you.
```

warning

```
1  g <- function() warning("Managed to recover.")
2
3  test_that("g() warns", {
4    expect_snapshot(g())
5  })
```

── Warning: g() warns
───

```
Adding new snapshot:
Code
  g()
Condition
  Warning in `g()`:
  Managed to recover.
```

> 💡 **Tip**
>
> Snapshot records both the *condition* and the corresponding *message.*
>
> You can now rest assured that the users are getting informed the way you want! 😌

Capturing errors

In case of an error, the function `expect_snapshot()` itself will produce an error.

You can capture it:

```
1  test_that("`log()` errors", {
2    expect_snapshot(log("x"), error = TRUE)
3  })
```

```
── Warning: `log()` errors ─────────────────────────────────────────
Adding new snapshot:
Code
  log("x")
Condition
  Error in `log()`:
  ! non-numeric argument to mathematical function
```

Further reading

- testthat article on snapshot testing

- Introduction to golden testing

- Docs for Jest library in JavaScript, which inspired snapshot testing implementation in testthat

Testing graphical outputs

To create graphical expectations, you will use testthat extension package: `{vdiffr}`.

How does `{vdiffr}` work?

vdiffr introduces `expect_doppelganger()` to generate testthat expectations for graphics. It does this by writing SVG snapshot files for outputs!

The figure to test can be:

- a `ggplot` object (from `ggplot2::ggplot()`)

- a `recordedplot` object (from `grDevices::recordPlot()`)

- any object with a `print()` method

> ⓘ **Note**
>
> - If test file is called `test-foo.R`, the snapshot will be saved to `test/testthat/_snaps/foo` folder.
>
> - In this folder, there will be one `.svg` file for every test in `test-foo.R`.
>
> - The name for the `.svg` file will be sanitized version of `title` argument to `expect_doppelganger()`.

Example function

Let's say you want to write a unit test for the following function:

Source code

```
1  library(ggplot2)
2
3  create_scatter <- function() {
4    ggplot(mtcars, aes(wt, mpg)) +
5      geom_point(size = 3, alpha = 0.75) +
6      geom_smooth(method = "lm")
7  }
```

Output

```
1  create_scatter()
```


Note that you want to test that the graphical output *looks* as expected, and this expectation is difficult to capture with a unit test.

Graphical snapshot test

You can use `expect_doppelganger()` from **vdiffr** to test this!

The *first time* you run the test, it'd generate an `.svg` file with expected output.

```
1  test_that("`create_scatter()` plots as expected", {
2    expect_doppelganger(
3      title = "create scatter",
4      fig = create_scatter(),
5    )
6  })
```

```
── Warning: `create_scatter()` plots as expected ──────────────────────────────
Adding new file snapshot: 'tests/testthat/_snaps/create-scatter.svg'
```

> ⚠️ **Warning**
>
> The first time a snapshot is created, it becomes *the truth* against which future function behaviour will be compared.
>
> Thus, it is **crucial** that you carefully check that the output is indeed as expected. 🔍
>
> You can open `.svg` snapshot files in a web browser for closer inspection.

What test success looks like

If you run the test again, it'll succeed:

```
1  test_that("`create_scatter()` plots as expected", {
2    expect_doppelganger(
3      title = "create scatter",
4      fig = create_scatter(),
5    )
6  })
```

Test passed 🌈

What test failure looks like

When function changes, snapshot doesn't match the reference, and the test fails:

Changes to function

```
1  create_scatter <- function() {
2    ggplot(mtcars, aes(wt, mpg)) +
3      geom_point(size = 2, alpha = 0.85) +
4      geom_smooth(method = "lm")
5  }
```

Test failure

```
1  test_that("`create_scatter()` plots as expected", {
2    expect_doppelganger(
3      title = "create scatter",
4      fig = create_scatter(),
5    )
6  })
7
8  ── Failure ('<text>:3'): `create_scatter()` plots as expected
9  Snapshot of `testcase` to 'slides.qmd/create-scatter.svg' has
10 Run `testthat::snapshot_review('slides.qmd/')` to review chang
11 Backtrace:
12  1. vdiffr::expect_doppelganger(...)
13  3. testthat::expect_snapshot_file(...)
14 Error in `reporter$stop_if_needed()`:
15 ! Test failed
```

Fixing tests

Running `snapshot_review()` launches a Shiny app which can be used to either accept or reject the new output(s).

💡 Why are my snapshots for plots failing?! 😔

If tests fail even if the function didn't change, it can be due to any of the following reasons:

- R's graphics engine changed
- ggplot2 itself changed
- non-deterministic behaviour
- changes in system libraries

For these reasons, snapshot tests for plots tend to be fragile and are not run on CRAN machines by default.

Further reading

- **vdiffr** package website

Testing entire files

Whole file snapshot testing makes sure that media, data frames, text files, etc. are as expected.

Writing test

Let's say you want to test JSON files generated by `jsonlite::write_json()`.

Test

```
 1  # File: tests/testthat/test-write-json.R
 2  test_that("json writer works", {
 3    r_to_json <- function(x) {
 4      path <- tempfile(fileext = ".json")
 5      jsonlite::write_json(x, path)
 6      path
 7    }
 8
 9    x <- list(1, list("x" = "a"))
10    expect_snapshot_file(r_to_json(x), "demo.json")
11  })
```

```
── Warning: json writer works
────────────────────────────────────────────────

Adding new file snapshot:
'tests/testthat/_snaps/demo.json'
```

Snapshot

```
 1  [[1],{"x":["a"]}]
```

> ⓘ **Note**
>
> - To snapshot a file, you need to write a helper function that provides its path.
>
> - If a test file is called `test-foo.R`, the snapshot will be saved to `test/testthat/_snaps/foo` folder.
>
> - In this folder, there will be one file (e.g. `.json`) for every `expect_snapshot_file()` expectation in `test-foo.R`.
>
> - The name for snapshot file is taken from `name` argument to `expect_snapshot_file()`.

What test success looks like

If you run the test again, it'll succeed:

```r
# File: tests/testthat/test-write-json.R
test_that("json writer works", {
  r_to_json <- function(x) {
    path <- tempfile(fileext = ".json")
    jsonlite::write_json(x, path)
    path
  }

  x <- list(1, list("x" = "a"))
  expect_snapshot_file(r_to_json(x), "demo.json")
})
```

Test passed 😀

What test failure looks like

If the new output doesn't match the expected one, the test will fail:

```r
1   # File: tests/testthat/test-write-json.R
2   test_that("json writer works", {
3     r_to_json <- function(x) {
4       path <- tempfile(fileext = ".json")
5       jsonlite::write_json(x, path)
6       path
7     }
8
9     x <- list(1, list("x" = "b"))
10    expect_snapshot_file(r_to_json(x), "demo.json")
11  })
```

```
── Failure: json writer works ──────────────────────────────
Snapshot of `r_to_json(x)` to 'slides.qmd/demo.json' has changed
Run `testthat::snapshot_review('slides.qmd/')` to review changes

Error:
! Test failed
```

Fixing tests

Running `snapshot_review()` launches a Shiny app which can be used to either accept or reject the new output(s).

Further reading

Documentation for `expect_snapshot_file()`

Testing Shiny applications

To write formal tests for Shiny applications, you will use
testthat extension package: `{shinytest2}`.

How does `{shinytest2}` work?

shinytest2 uses a Shiny app (how meta! 😅) to record user interactions with the app and generate snapshots of the application's state. Future behaviour of the app will be compared against these snapshots to check for any changes.

Exactly how tests for Shiny apps in R package are written depends on how the app is stored. There are two possibilities, and you will discuss them both separately.

Stored in `/inst` folder

```
├── DESCRIPTION
├── R
├── inst
│   └── sample_app
│       └── app.R
```

Returned by a function

```
├── DESCRIPTION
├── R
│   └── app-function.R
```

Shiny app in subdirectory

```
├── DESCRIPTION
├── R
├── inst
│   └── sample_app
│       └── app.R
```

Example app

Let's say this app resides in the `inst/unitConverter/app.R` file.

App

Code

```
 1  library(shiny)
 2
 3  ui <- fluidPage(
 4    titlePanel("Convert kilograms to grams"),
 5    numericInput("kg", "Weight (in kg)", value = 0),
 6    textOutput("g")
 7  )
 8
 9  server <- function(input, output, session) {
10    output$g <- renderText(
11      paste0("Weight (in g): ", input$kg * 1000)
12    )
13  }
14
15  shinyApp(ui, server)
```

Generating a test

To create a snapshot test, go to the app directory and run `record_test()`.

Auto-generated artifacts

Test

```
1  library(shinytest2)
2
3  test_that("{shinytest2} recording: unitConverter", {
4    app <- AppDriver$new(
5    name = "unitConverter", height = 543, width = 426)
6    app$set_inputs(kg = 1)
7    app$set_inputs(kg = 10)
8    app$expect_values()
9  })
```

Snapshot

Convert kilograms to grams

Weight (in kg)

10

Weight (in g): 10000

> ## ⓘ Note
>
> - `record_test()` will auto-generate a test file in the app directory. The test script will be saved in a *subdirectory* of the app (`inst/my-app/tests/testthat/test-shinytest2.R`).
>
> - There will be one `/tests` folder inside *every* app folder.
>
> - The snapshots are saved as `.png` file in `tests/testthat/test-shinytest2/_snaps/{.variant}/shinytest2`. The `{.variant}` here corresponds to operating system and R version used to record tests. For example, `_snaps/windows-4.1/shinytest2`.

Creating a driver script

Note that currently your test scripts and results are in the `/inst` folder, but you'd also want to run these tests automatically using testthat.

For this, you will need to write a driver script like the following:

```
1  library(shinytest2)
2
3  test_that("`unitConverter` app works", {
4    appdir <- system.file(package = "package_name", "unitConverter")
5    test_app(appdir)
6  })
```

Now the Shiny apps will be tested with the rest of the source code in the package! 🎊

> 💡 **Tip**
>
> You save the driver test in the `/tests` folder (`tests/testthat/test-inst-apps.R`), alongside other tests.

What test failure looks like

Let's say, while updating the app, you make a mistake, which leads to a failed test.

Changed code with mistake

```
1  ui <- fluidPage(
2    titlePanel("Convert kilograms to grams"),
3    numericInput("kg", "Weight (in kg)", value = 0),
4    textOutput("g")
5  )
6
7  server <- function(input, output, session) {
8    output$g <- renderText(
9      paste0("Weight (in kg): ", input$kg * 1000) # should be `"Weight (in g): "`
10   )
11 }
12
13 shinyApp(ui, server)
```

Test failure JSON diff

```
1  Diff in snapshot file `shinytest2unitConverter-001.json`
2  < before                                   > after
3  @@ 4,5 @@                                   @@ 4,5 @@
4      },                                          },
5      "output": {                                 "output": {
6  <      "g": "Weight (in g): 10000"   >         "g": "Weight (in kg): 10000"
7      },                                          },
8      "export": {                                 "export": {
```

Updating snapshots

Fixing this test will be similar to fixing any other snapshot test you've seen thus far.

`{testthat2}` provides a Shiny app for comparing the old and new snapshots.

Function returns Shiny app

```
├── DESCRIPTION
├── R
│   └── app-function.R
```

Example app and test

The only difference in testing workflow when Shiny app objects are created by functions is that you will write the test ourselves, instead of shinytest2 auto-generating it.

Source code

```
 1  # File: R/unit-converter.R
 2  unitConverter <- function() {
 3    ui <- fluidPage(
 4      titlePanel("Convert kilograms to grams"),
 5      numericInput("kg", "Weight (in kg)", value = 0),
 6      textOutput("g")
 7    )
 8
 9    server <- function(input, output, session) {
10      output$g <- renderText(
11        paste0("Weight (in g): ", input$kg * 1000)
12      )
13    }
14
15    shinyApp(ui, server)
16  }
```

Test file to modify

```
 1  # File: tests/testthat/test-unit-converter.R
 2  test_that("unitConverter app works", {
 3    shiny_app <- unitConverter()
 4    app <- AppDriver$new(shiny_app)
 5  })
```

Generating test and snapshots

you call `record_test()` directly on a Shiny app object, copy-paste commands to the test script, and run `devtools::test_active_file()` to generate snapshots.

```
R  unit-converter.R  ×        R  test-unit-converter.R  ×

1  test_that("unitConverter app works", {
2    shiny_app <- unitConverter()
3    app <- AppDriver$new(shiny_app)
4  })
5

2:3    test_that("unitConverter app works")                    R Script

Console    Terminal  ×    Background Jobs  ×

R 4.2.1 · ~/Documents/GitHub/minimalReprex/
>

▶  0:00 / 0:39
```
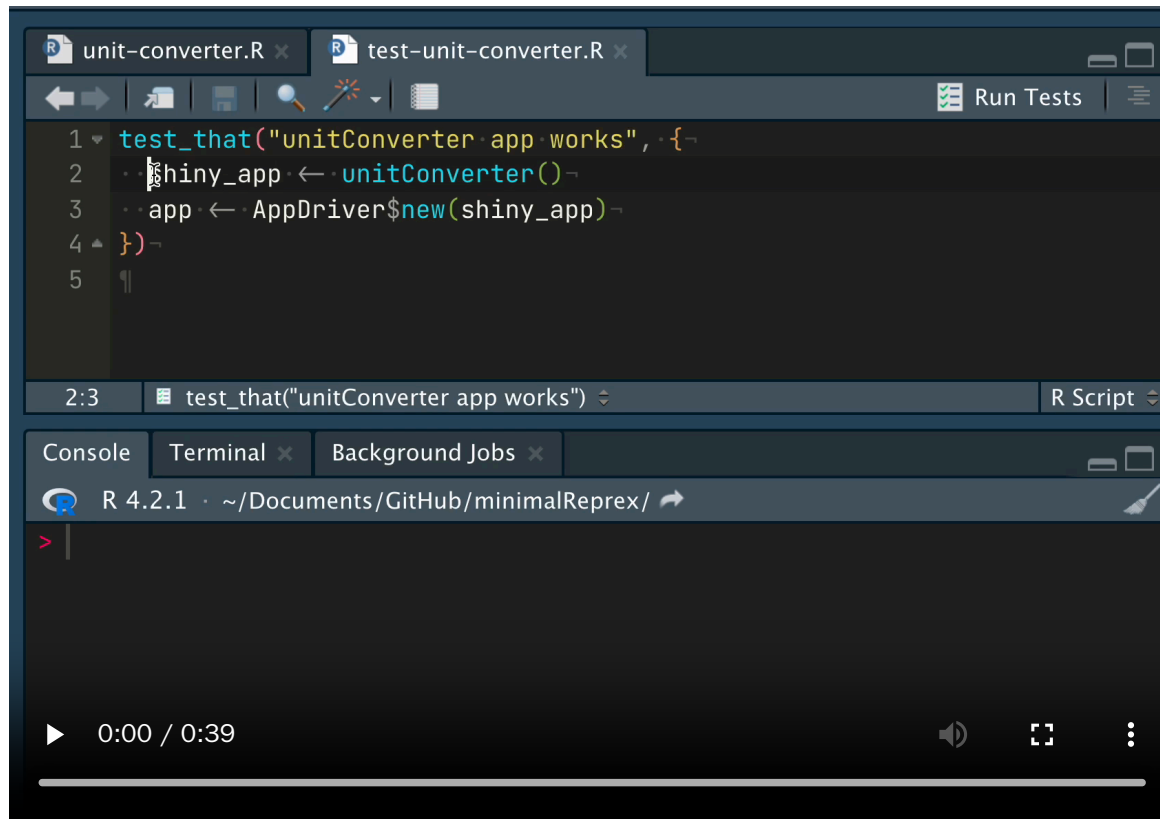
Testing apps from frameworks

This testing workflow is also relevant for app frameworks (e.g. `{golem}`, `{rhino}`, etc.).

golem

Function in `run_app.R` returns app.

```
├── DESCRIPTION
├── NAMESPACE
├── R
│   ├── app_config.R
│   ├── app_server.R
│   ├── app_ui.R
│   └── run_app.R
```

rhino

Function in `app.R` returns app.

```
├── app
│   ├── js
│   │   └── index.js
│   ├── logic
│   │   └── __init__.R
│   ├── static
│   │   └── favicon.ico
│   ├── styles
│   │   └── main.scss
│   ├── view
│   │   └── __init__.R
│   └── main.R
├── tests
│   ├── ...
├── app.R
├── RhinoApplication.Rproj
├── dependencies.R
├── renv.lock
└── rhino.yml
```

Final directory structure

The final location of the tests and snapshots should look like the following for the two possible ways Shiny apps are included in R packages.

Stored in `/inst` folder

```
├── DESCRIPTION
├── R
├── inst
│   └── sample_app
│       ├── app.R
│       └── tests
│           ├── testthat
│           │   ├── _snaps
│           │   │   └── shinytest2
│           │   │       └── 001.json
│           │   └── test-shinytest2.R
│           └── testthat.R
└── tests
    ├── testthat
    │   └── test-inst-apps.R
    └── testthat.R
```

Returned by a function

```
├── DESCRIPTION
├── R
│   └── app-function.R
└── tests
    ├── testthat
    │   ├── _snaps
    │   │   └── app-function
    │   │       └── 001.json
    │   └── test-app-function.R
    └── testthat.R
```

Testing multiple apps

For the sake of completeness, here is what the test directory structure would like when there are multiple apps in a single package.

Stored in `/inst` folder

```
├── DESCRIPTION
├── R
├── inst
│   └── sample_app1
│       ├── app.R
│       └── tests
│           ├── testthat
│           │   ├── _snaps
│           │   │   └── shinytest2
│           │   │       └── 001.json
│           │   └── test-shinytest2.R
│           └── testthat.R
│   └── sample_app2
│       ├── app.R
│       └── tests
│           ├── testthat
│           │   ├── _snaps
│           │   │   └── shinytest2
│           │   │       └── 001.json
│           │   └── test-shinytest2.R
│   │       │
```

Returned by a function

```
├── DESCRIPTION
├── R
│   └── app-function1.R
│   └── app-function2.R
└── tests
    ├── testthat
    │   ├── _snaps
    │   │   └── app-function1
    │   │       └── 001.json
    │   │   └── app-function2
    │   │       └── 001.json
    │   └── test-app-function1.R
    │   └── test-app-function2.R
    └── testthat.R
```

Advanced topics

The following are some advanced topics that are beyond the scope of the current presentation, but you may wish to know more about.

> ○ **Extra**
>
> - If you want to test Shiny apps with continuous integration using shinytest2, read this article.
>
> - shinytest2 is a successor to shinytest package. If you want to migrate from the latter to the former, have a look at this.

Further reading

- Testing chapter from *Mastering Shiny* book
- shinytest2 article introducing its workflow
- shinytest2 article on how to test apps in R packages

Headaches

It's not all kittens and roses when it comes to snapshot testing.

Let's see some issues you might run into while using them. 🤕

Testing behavior that you don't own

Let's say you write a graphical snapshot test for a function that produces a `ggplot` object. If ggplot2 authors make some modifications to this object, your tests will fail, even though your function works as expected!

In other words, *your* tests are now at the mercy of *other package authors* because snapshots are capturing things beyond your package's control.

> ⚠️ **Caution**
>
> Tests that fail for reasons other than what they are testing for are problematic. Thus, be careful about *what* you snapshot and keep in mind the maintenance burden that comes with dependencies with volatile APIs.

> ℹ️ **Note**
>
> *A* way to reduce the burden of keeping snapshots up-to-date is to automate this process. But there is no free lunch in this universe, and now you need to maintain this automation! 🤷

Failures in non-interactive environments

If snapshots fail locally, you can just run `snapshot_review()`, but what if they fail in non-interactive environments (on CI/CD platforms, during `R CMD Check`, etc.)?

The easiest solution is to copy the new snapshots to the local folder and run `snapshot_review()`.

> 💡 **Tip**
>
> If expected snapshot is called (e.g.) `foo.svg`, there will be a new snapshot file `foo.new.svg` in the same folder when the test fails.
>
> `snapshot_review()` compares these files to reveal how the outputs have changed.

But where can you find the new snapshots?

Accessing new snapshots

In local `R CMD Check`, you can find new snapshots in `.Rcheck` folder:

```
1  package_name.Rcheck/tests/testthat/_snaps/
```

On GitHub actions, you need to upload snapshots:

```
1        - uses: r-lib/actions/check-r-package@v2
2          with:
3            upload-snapshots: true
```

Code review with snapshot tests

Despite snapshot tests making the expected outputs more human-readable, given a big enough change and complex enough output, sometimes it can be challenging to review changes to snapshots.

How do you review pull requests with complex snapshots changes?

Use tools provided for code review by hosting platforms (like GitHub). For example, to review changes in SVG snapshots:

Danger of silent failures

Given their fragile nature, snapshot tests are skipped on CRAN by default.

Although this makes sense, it means that you miss out on anything but a breaking change from upstream dependency. E.g., if ggplot2 (hypothetically) changes how the points look, you won't know about this change until you *happen to* run your snapshot tests again locally or on CI/CD.

Unit tests run on CRAN, on the other hand, will fail and you will be immediately informed about it.

> 💡 **Tip**
>
> A way to insure against such silent failures is to run tests **daily** on CI/CD platforms (e.g. nightly builds).

Parting wisdom

What *not* to do

Don't use snapshot tests for *everything*

It is tempting to use them everywhere out of laziness. But they are sometimes inappropriate (e.g. when testing requires external benchmarking).

Let's say you write a function to extract estimates from a regression model.

```r
1  extract_est <- function(m) m$coefficients
```

Its test should compare results against an external benchmark, and not a snapshot.

Good

```r
1  test_that("`extract_est()` works", {
2    m <- lm(wt ~ mpg, mtcars)
3    expect_equal(
4      extract_est(m)[[1]],
5      m$coefficients[[1]]
6    )
7  })
```

Bad

```r
1  test_that("`extract_est()` works", {
2    m <- lm(wt ~ mpg, mtcars)
3    expect_snapshot(extract_est(m))
4  })
```

Snapshot for humans, not machines

Snapshot testing is appropriate when the human needs to be in the loop to make sure that things are working as expected. Therefore, the snapshots should be human readable.

E.g. if you write a function that plots something:

```
1  point_plotter <- function() ggplot(mtcars, aes(wt, mpg)) + geom_point()
```

To test it, you should snapshot the *plot*, and not the underlying *data*, which is hard to make sense of for a human.

Good

```
1  test_that("`point_plotter()` works", {
2    expect_doppelganger(
3      "point-plotter-mtcars",
4      point_plotter()
5    )
6  })
```

Bad

```
1  test_that("`point_plotter()` works", {
2    p <- point_plotter()
3    pb <- ggplot_build(p)
4    expect_snapshot(pb$data)
5  })
```

Don't blindly accept snapshot changes

Resist formation of such a habit.

testthat provides tools to make it very easy to review changes, so no excuses!

Self-study

In this presentation, you deliberately kept the examples and the tests simple.

To see a more realistic usage of snapshot tests, you can study open-source test suites.

Suggested repositories

Print outputs

- `{cli}` (for testing command line interfaces)

- `{pkgdown}` (for testing generated HTML documents)

- `{dbplyr}` (for testing printing of generated SQL queries)

- `{gt}` (for testing table printing)

Visualizations

- `{ggplot2}`

- `{ggstatsplot}`

Shiny apps

- `{shinytest2}`

- `{designer}`

Activities

If you feel confident enough to contribute to open-source projects to practice these skills, here are some options.

Practice makes it perfect

These are only suggestions. Feel free to contribute to any project you like! 🤝

> 💡 **Suggestions**
>
> - See if ggplot2 extensions you like use snapshot tests for graphics. If not, you can add them for key functions.
>
> - Check out hard reverse dependencies of `{shiny}`, and add snapshot tests using shinytest2 to an app of your liking.
>
> - Add more vdiffr snapshot tests to plotting functions in `{see}`, a library for statistical visualizations (I can chaperone your PRs here).
>
> - shinytest2 is the successor to shinytest package. Check out which packages currently use it for testing Shiny apps, and see if you can use shinytest2 instead (see how-to here).

For more

For a comprehensive collection of packages for unit testing in R, see this page.

If you are interested in good programming and software development practices, check out my other slide decks.

Find me at...

 Twitter

 LikedIn

 GitHub

 Website

 E-mail

Thank You

And Happy Snapshotting! 😊

Session information

```
1  sessioninfo::session_info(include_base = TRUE)
```

```
— Session info ——————————————————————————————————————————————————————————
 setting  value
 version  R version 4.5.1 (2025-06-13)
 os       Ubuntu 24.04.2 LTS
 system   x86_64, linux-gnu
 ui       X11
 language (EN)
 collate  C.UTF-8
 ctype    C.UTF-8
 tz       UTC
 date     2025-06-22
 pandoc   3.7.0.2 @ /opt/hostedtoolcache/pandoc/3.7.0.2/x64/ (via rmarkdown)
 quarto   1.8.14 @ /usr/local/bin/quarto

— Packages ———————————————————————————————————————————————————————————————
 package      * version date (UTC) lib source
 base         * 4.5.1   2025-06-13 [3] local
 brio           1.1.5   2024-04-24 [1] RSPM
 cli            3.6.5   2025-04-23 [1] RSPM
 compiler       4.5.1   2025-06-13 [3] local
```