Chapter 4
Testing

4.1 Overview

Between all of the profiling and load testing, we have made a number of changes to our shiny application since the first version that we started with. But can we be certain that the app is doing the same thing now (although much faster) as it was when we started out?

One of the first things that we should have done, before we made any changes, was generate a suite of tests that could be re-run regularly, to ensure we were not impacting the functionality of our application.

4.2 Testing Shiny Apps

We want to ensure our apps continue to work as we add or update features, or other changes happen such as a new R version or new versions of packages are deployed. Until recently, testing a shiny app has meant manually testing things out through a browser, which takes a long time and can be laborious. We can now automate this process using the **shinytest** package.

The **shinytest** package allows us to carry out the process of manually testing our app just once while our app interactions are recorded automatically in a test script. When we want to re-test our app we can simply re-run the script to simulate our interactions and ensure our app's behaviour has not changed.

4.2.1 Getting Started

The first step is to install **shinytest**. We can do this as usual from CRAN, but shinytest also needs a "headless browser" to be installed to simulate the web browsing session without bringing up a fully interactive web browser. We can do this once **shinytest** is installed as follows:

```
install.packages("shinytest")
shinytest::installDependencies()
```

The workflow is then as follows:

- Start recording
- Manually test our app
- Take "snapshots" of our app's state at key points
- Stop recording

4.3 Recording our First Test

As an example we will work on the most efficient version of our application, which includes plot caching, but remember that the full range of improvements will be less obvious when running an individual session.

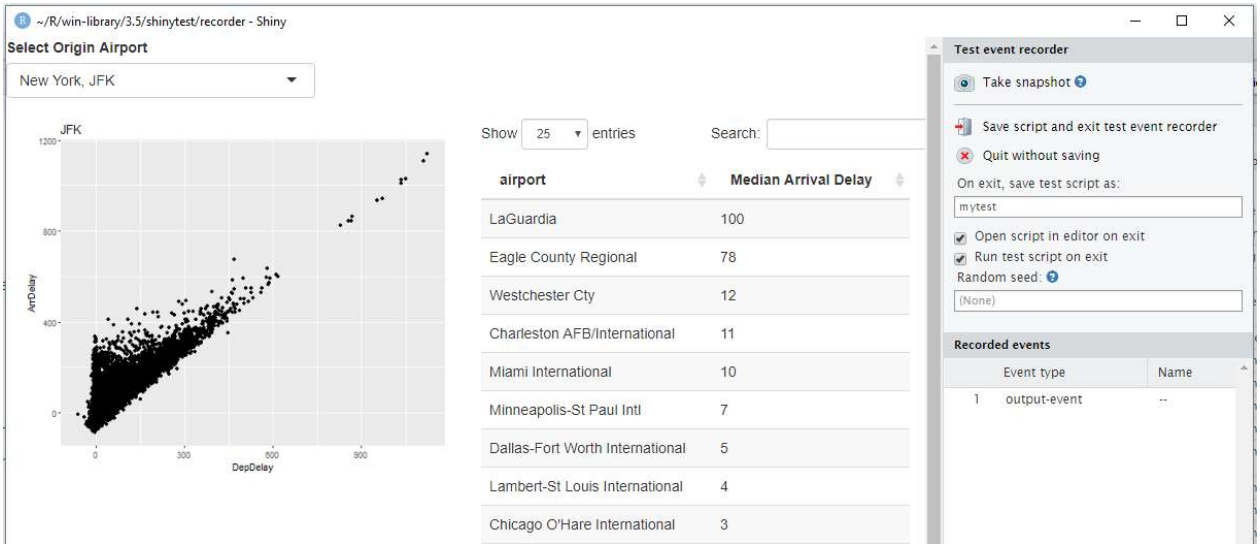
We will run `recordTest` on our app and take a few snapshots. Our manual test script (for a person to follow) might look like this:

- Run the app
- Check if the start screen looks correct (where "New York, JFK" is the selected airport)
- Change the Airport to "San Francisco Int."
- Check that the plot and table updated

We'll do a similar thing, taking a snapshot whenever we make a visual check.

```
> library(shinytest)
> recordTest(app = "app3_plotcache")
```

The app will start with the addition of a "Test event recorder" tool.



From here we can "Take a snapshot" each time we make a check, and carry out the test script. As we interact with the app the list of "Recorded events" will grow.

Once we have finished we can click "Save script and exit test event recorder". RStudio will save our test script under a new folder within our app called `tests`, along with the

snapshots we took during our tests. The console output displays exactly where the tests are saved:

```
> recordTest(app = "app3_plotcache")

Listening on http://127.0.0.1:7087
Saved test code to app3_plotcache/tests/change_airport.R
Running change_airport.R

==== Comparing change_airport...
  No existing snapshots at change_airport-expected/. This is a first
run of tests.

Updating baseline results at
app3_plotcache/tests/change_airport-expected...
Renaming app3_plotcache/tests/change_airport-current
      => app3_plotcache/tests/change_airport-expected.
```

The script records the actions we took as part of our testing, the snapshots record the state of our app at our chosen points in time (which we would have verified to be correct when we did our manual test). We can now re-run our test script and shinytest will verify our app is in the correct state throughout the test by comparing it with the snapshot.

4.3.1 Re-Running the Test

Now that our test is recorded we never need to manually do that test again! To get shinytest to reproduce the test we can call `testApp`.

```
> testApp(appDir = "app3_plotcache")
Running change_airport.R
==== Comparing change_airport... No changes.
```

The `testApp` function starts the app, carries out all of the interactions, and checks to ensure the app's state matches the snapshots at each point. In this case our test passed, so no further action is required.

4.3.2 Simulating a Test Failure

In reality our tests might fail after we inadvertently break something whilst adding another feature, or perhaps a new package version causing something in our app to work differently. To demonstrate this we will simulate a change to the app by changing the code.

Let's suppose we have made a mistake while updating out app, and the underlying data is now for a different airport to the one we are intending to show. In this case, we will change our input options so that when selecting "San Francisco Int", the data is in fact for Seattle.

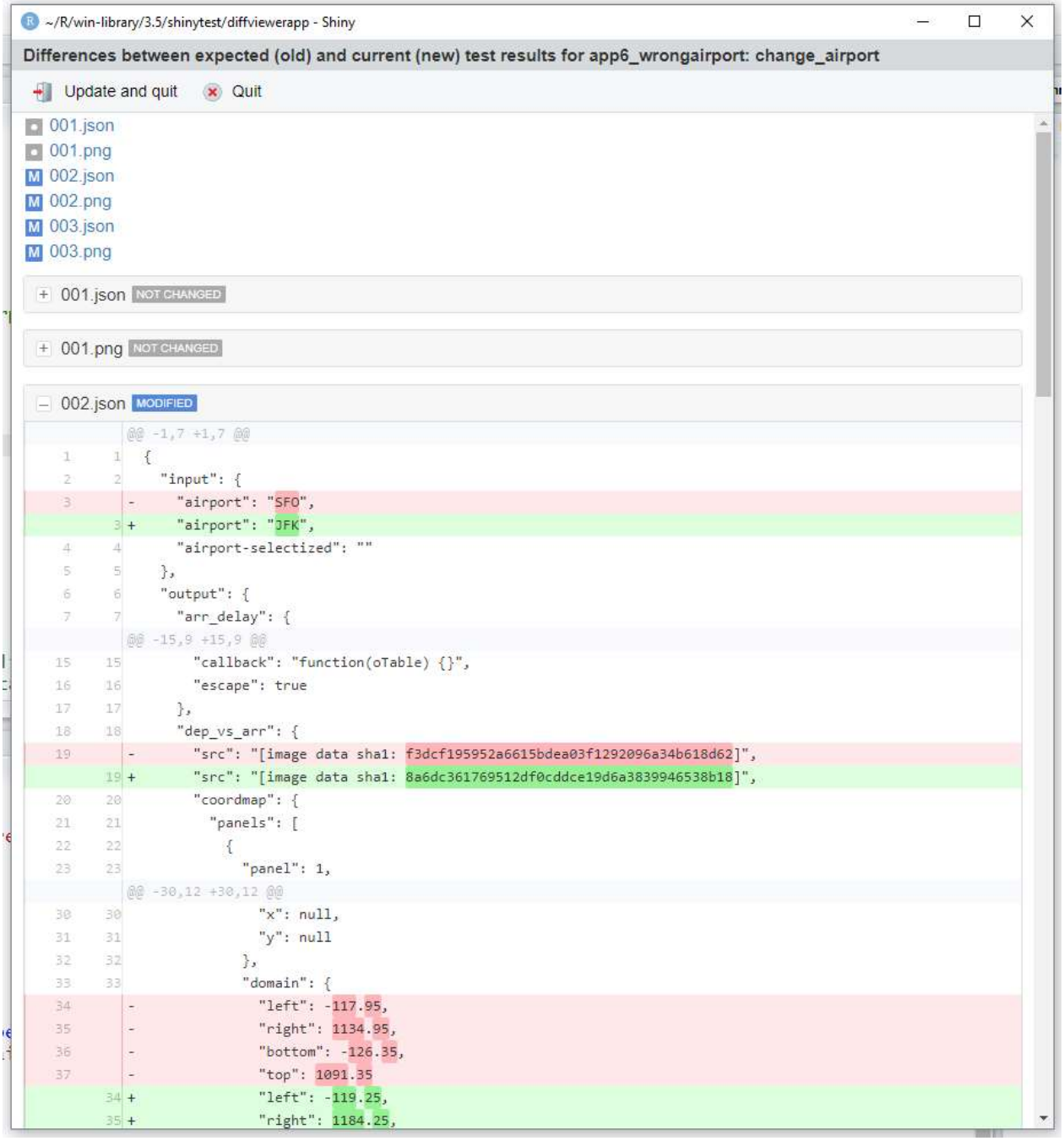
```
## SEA should actually be SFO
selectInput("airport", "Select Origin Airport",
                      choices = list("New York, JFK" = "JFK",
                                     "San Francisco Int." = "SEA",
                                     "Chicago O'Hare" = "ORD"))
```

Then we re-run the test script:

```
> testApp(appDir = "app6_wrongairport")
Running change_airport.R
==== Comparing mytest...
Differences detected between change_airport-current/ and
change_airport-expected/:

    Name         Status
    001.json     No change
    001.png      No change
    002.json  != Files differ
    002.png   != Files differ
    003.json  != Files differ
    003.png   != Files differ
Would you like to view the differences between expected and current
results [y/n]?
```

The console printout shows that the second two snapshots have changed but the first still passes. We can enter "y" to open a viewer to visualise the changes.



We can see how each snapshot has two representations: a JSON file containing structured data about the app's status, and a PNG file that is a screenshot of the app.

For the first failure, we can see in the JSON file that the previously selected airport was SFO but it is now JFK (because SFO wasn't an option the menu option wasn't changed). We can inspect the image differences in the lower pane, where changes are highlighted in red.

4.3.2.1 Responding to a Failed Test

The differences viewer prompts us with two alternatives:

1. Update and quit
2. Quit

If the changes we observe are not actually a problem, for example if an updated library changes the layout of a plot slightly, but it is still correct, we should choose "Update and quit". This will update the snapshots to reflect the app's new state. Subsequent runs of the test will then pass.

If the changes we observe are a genuine problem, we should chose Quit and fix the problem before re-running the test.

4.3.3 Editing Tests

The test script created by the test recorder is an R script that can be edited in the usual way. For example, the test script we recorded previously looks as follows:

```
app <- ShinyDriver$new("../")
app$snapshotInit("change_airport")

app$snapshot()
app$setInputs(airport = "SFO")
app$snapshot()
app$snapshot()
```

Scripts typically consist of an initialisation step where a shiny "driver" is set up and initialised, followed by a series of snapshots taken using `app$snapshot`, interspersed with scripted modifications of shiny input values using `app$setInputs`.

The test recorder will often capture superfluous user interactions, for example where we have slowly moved an input slider during our recording and multiple values have been recorded. For a faster test script we could edit the script to remove these unnecessary calls to `setInputs`.

To set multiple inputs at once we can provide several named arguments in a single call to `setInputs`, for example:

```
app$setInputs(mag = c(4.5, 5.5), stations = 50)
```

1. Using the exercise application, create a test script from the following workflow, taking a snapshot of the application after each step:
   a. Launch the application and allow to load
   b. Change the model coefficient to "DepTime"
   c. Change the number of samples to 17,000
   d. Change the model coefficient to "AirTime"
   e. Set the samples back to 10,000
   f. Set the coefficient back to "Distance"
2. Save the test script and execute the script to create a baseline
3. Run the tests again. Thinking about how the app runs and considering where the test failures are, what is the problem with these tests?
4. Re-create your test script, what option in the test recorder should you set to overcome the previous issue?
5. Ensure you have set the correct options and save your test script
6. If you re-run your new tests, are the results now as expected?