# Test Driven Development

I'd like to share this out with the team to start discussion and awareness of testing in spa apps.  Done correctly, it speeds development rather than slowing it. and of course, it’s invaluable during maintenance enhancements and adjustments.

<https://github.com/keslavi/react-16-seed-js> for the final test example, just showing the process here:

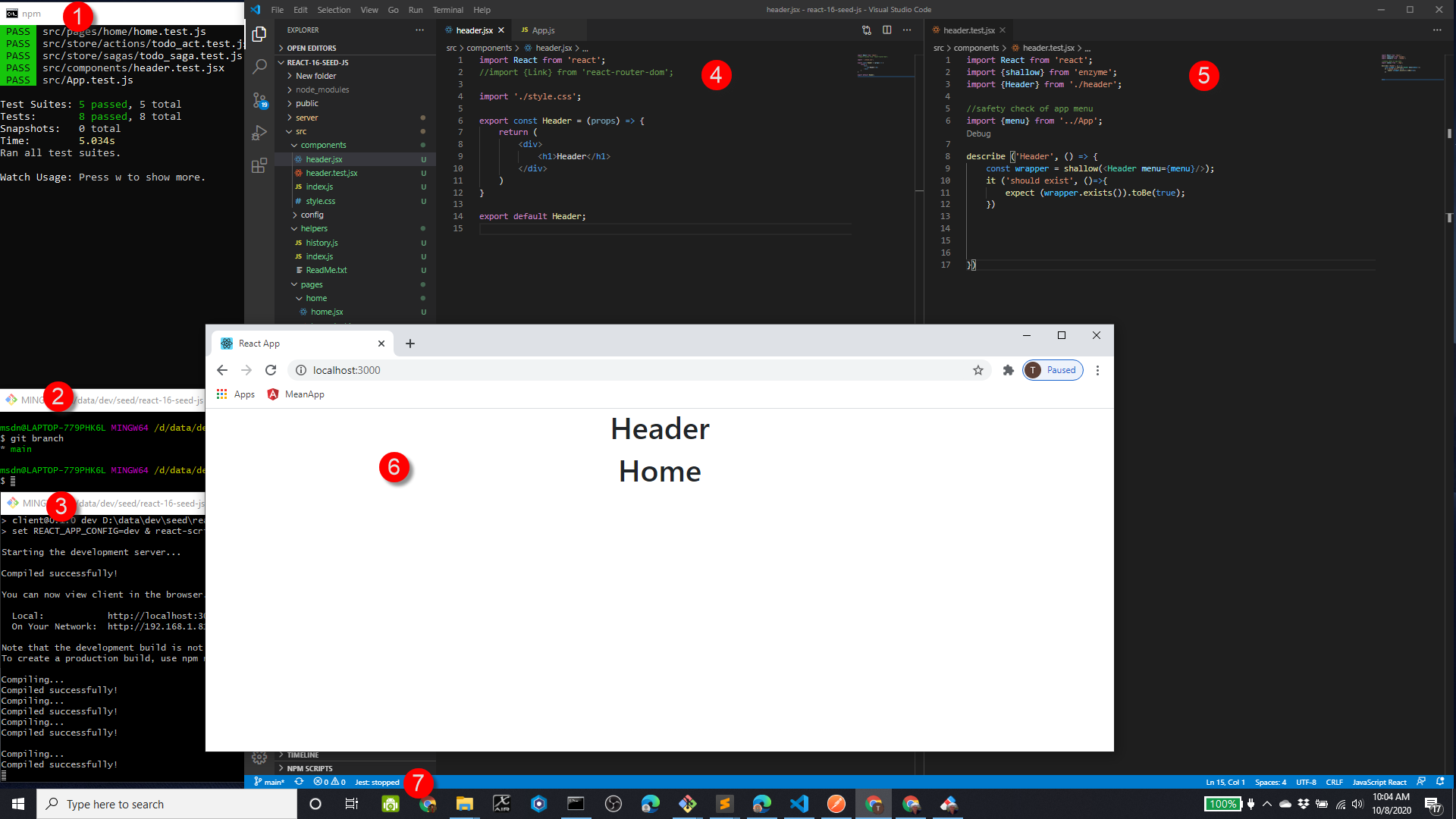
I'm switching the seed project from typescript back to javascript (ES6)  
  
in redoing this, I'm also redoing the unit tests that are run to ensure things are working as expected.   
  
I thought y'all might find this example of test driven development useful.   
  
I tend to use git bash and command windows over the VSCode terminal because it:

* increases the editor real estate for looking at code
* MULTIPLE terminals are being used (explained below)
* gives me real time feed back... sometimes VS code's terminal window is closed or lags/doesn't show as much info or color coding
* by having the IDE on the left, I can overlap but still see when a terminal window begins fussing at me, then navigate to the specific terminal.

OSX, Linux have way better terminals than Windows, but this is the best compromise I can get.  
  
Having said that, here's a breakdown of what's on the screen… example on next page:

1. terminal that runs the tests continuously. note that command window is used due to the better color coding when an error occurs.
2. terminal for doing git, npm operations.  using bash because of:
   1. coloring.
   2. branch name is displayed
   3. $ cursor is better viewed than windows LOOOONGPATH>
3. Where run npm start provides feedback as app recompiles. git bash used again
4. VS Code with editor split layout.  
    (4= the actual code being worked on.)
5. the test file.
6. the browser displaying the app. usually on a different screen when available (remote desktop/display/ use all monitors for session)

**Example1: standard display**



this setup allows me to:

1. create the test... it ('should whatever')... jest extensions mean i can type "it", arrow down and hit tab to autocomplete the boilerplate
2. watch the new test fail
3. implement whatever piece I'm building
4. watch the test pass
5. view the app to visually see my changes

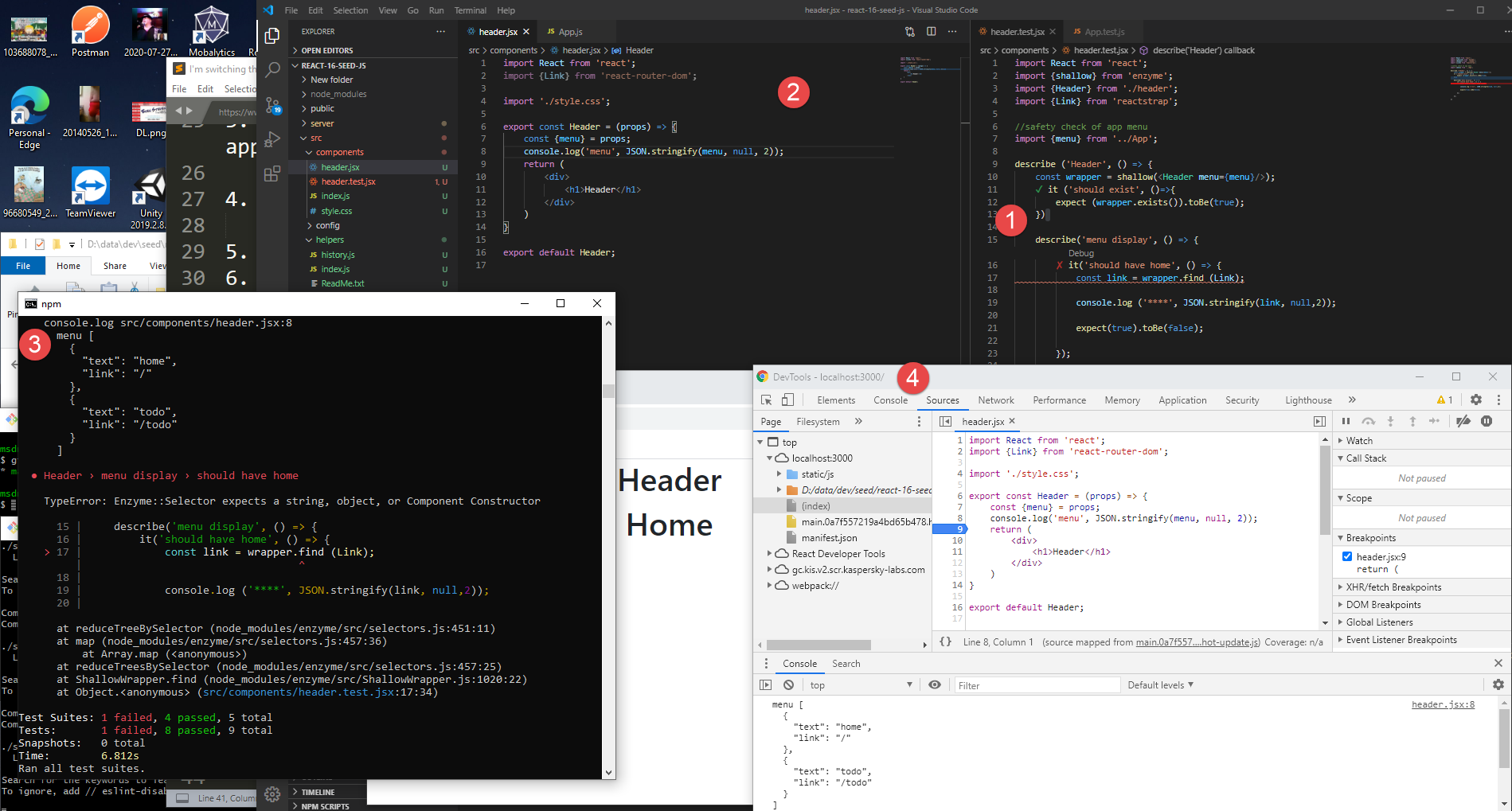
**Example 2: Creating a new test and functionality**

below, I'm adding functionality to accept and build a menu dynamically in the header based on a json object.

step 1: I wrap the overall functionality in a describes statement and create my first "it" test which fails.   I have a general idea of how to accomplish my functionality, but no specifics yet. I need to do a combination of coding the component, watching the browser debug window, and eventually shape the test appropriately.  screens sections as follows:

1. where I'm writing the test.  The new test is created first and fails.  note that Visual code is giving me check marks and X marks to indicate passing and failing.
2. where I'm writing the code to accomplish my functionality.
3. example of why i use the command prompt to display test results... it will provide feedback to help me complete both the code and the test.
4. using the browser debug window to get live results back and pause the code to debug.  This frequently allows me to verify structures and potentially copy out to coding window find and copy syntax into the test.

(example on next page)



**Final Result**:

Here’s the final result.   It took more than a few console.log statements go get the right syntax, and converting wrapper to a json string to analyze doesn’t work at all.

Lots of refactoring and debugging to get to the final, clean test.

Used this reference:

<https://enzymejs.github.io/enzyme/docs/api/ReactWrapper/find.html>

and kept working with it till I got the right combination.  It’s not too hard, but the first few times can be a doozy till you remember the syntax.

In the it statement was doing things like:

Const gah= wrapper.find(‘a’);

Console.log (‘\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*’);

Console.log(‘items’, gah.count);

Console.log (‘\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*’);

