Practical: Testing and Linting

Initial Due Date: 2023-02-23 8:15AM
Final Due Date: 2023-03-31 4:15PM

Goals

- Implement unit tests
- Use a linter to write more consistent, more maintainable code

In the last practical, you created your first npm package. Today, we will actually use the module we installed (Jest) to do some unit testing and we will add a linter called ESLint to perform static analysis.

I encourage you to treat this (and other practical exercises) as a tutorial that you are trying to learn from, rather than an assignment you are trying to complete as quick as possible. Take your time, read the notes thoroughly and don't hesitate to ask questions.

Prerequisite

1. Create the git repository for your practical by accepting the assignment from GitHub Classroom. This will create a new repository for you with a skeleton npm package already set up for you.
2. Clone the repository to you computer with 💻 `git clone` (get the name of the repository from GitHub).
3. Open up the `package.json` file and add your name as the author of the package and the URL of your git repository.

Setting up unit testing

We want to add automated unit tests for our functions. Unit testing typically requires 1) a *test runner* to automatically run all the tests and report the results, and 2) an *assertion library* for implementing expectations about the behavior of the code under test. We will use the Jest unit testing package, which provides both. Jest is one of many possible unit testing libraries; it is not necessarily the best (a matter of opinion) but is widely used and integrated into a number of tools we use throughout the semester.

Install Jest by running 💻 `npm install -D jest @swc/jest` in the shell.

As a reminder, the `-D` option specifies that you want to update `package.json` with this dependency, and that it is a "development" dependency. You only need Jest when developing this module (when you would run the tests) and not in production. The second package is a tool for transpiling JavaScript so that we can use a consistent, modern, feature set across our projects regardless of the target. In this case, we want to use ES modules with are only partially supported Jest. The latter tool helps us work around that limitation. It is inserted into the workflow by the configuration in the `jest.config.ts` file in the skeleton.

Notice that the `package.json` file now specifies this new dependency (your version for this package and others may be slightly different):

```
"devDependencies": {
  "@swc/jest": "^0.2.24",
```

```
    "jest": "^29.0.2"
  }
```

Now that you have a testing library, you want to update the "test" script specified in the `package.json` file to run Jest. To do so, edit your `package.json` file to include:

```
"scripts": {
  "test": "jest"
},
```

You can now run Jest with 💻 `npm test` or 💻 `npm run test`. However, since you don't have any tests yet, you will get an error.

Example: validSong()

We are going to write a function to check if a string contains a song (or at the very least, a valid sequence of notes). Our song will be defined as a string containing notes separated by spaces. The notes will be a single upper case letter in the set [A-G]. They can optionally be modified by a sharp ('#') or a flat ('b'). For reasons we will not get into, there is no 'B#', 'Cb', 'E#' or 'Fb'.

```
> validSong("C C G G A A G")
true
> validSong("Ab Bb C C# Db D  G")
true
> validSong("Ab Cb")
false
> validSong("Ab Z")
false
```

Writing tests

We are going to practice test driven development (TDD) to create this function.

In `index.js`, put in the function declaration, with no body other than a simple return statement.

```
const validSong = (song) => {

  return false;
}

export default validSong;
```

Copy

Now we are going to pick one feature of this function, and write a test to test it. We will start by testing that it accepts valid, unmodified notes.

Open the file called `index.test.js` (jest will automatically run any files ending in `test.js`) and create a new test suite and a test.

Jest provides the `test(string, fn)` function. This is a basic test comprising a string description that will be printed when the test is run and a function that will provide the body of the test. We have wrapped that test in the `describe` function, which helps group tests that share common setup or teardown (described more below).

The `test` function should contain one or more assertions, i.e., tests of state or values in your code. The `expect(value)` function takes as an argument a value generated by your code in some way and returns an "expectation object". To turn this into a test, you apply a matcher to test that value. There are a number of different matchers. Jest will run all of your tests for you and keep track of how many tests pass and how many fail.

You can have multiple assertions within a single test function. All of the assertions should contribute in some way to the test.

Jest provides another function named `describe`, which allows us to wrap multiple tests together into a "suite". These tests can be loosely coupled. Perhaps they all test the same component or approach testing a function from different directions. Often the tests in a single `describe` all share common setup and tear down functionality, that is they all need the same work to be performed before the test is run and after the test is complete (e.g., to make the tests repeatable and independent).

Paste the following into `index.test.js` to get started.

```
import validSong from "./index";

describe("Testing validSong()", ()=>{
  test("validSong: accepts valid notes", ()=>{

  });
});
```

Copy

Now we need an assertion, which has `expect` and a matcher. Jest has a lot of matchers, but we can stick with `toBeTruthy` and `toBeFalsy` since our function returns a Boolean value.

Since we have a fixed number of valid notes, we can check them all. Add this to the body of your test:

```
const validNotes = ["A","B","C","D","E","F","G"];

validNotes.forEach((note)=>{
  expect(validSong(note)).toBeTruthy();
});
```

Copy

Note: Using a `forEach` to iterate over test cases should not be your first solution. It is appropriate here because we have a collection of identical tests to run.

These assertions only test if the function accepts single notes, we also need to make sure that it can handle strings with multiple notes.

Add a second test and name it "validSong: accepts compound strings". Test the function on the string "A B C D E F G".

Run the tests with `npm test`. They should fail since we have not yet implemented the function.

Satisfying tests

Now, we need to write the minimal amount of code to make sure that our tests pass. In this case, we can do that by changing the return value of the function to `true`.

Run the tests again. They should now pass.

Iterate

Clearly these were insufficient tests of the behavior we were targeting. When we specified that the function accepted a certain set of letters as valid notes, we really mean "uniquely". So, we need to test the "sad path" as well, what happens when we give the function invalid input.

We could test all other letters (or symbols!), but that is on the verge of overkill. At a certain point, we need to acknowledge that we are probably not learning more and we are just wasting time (remember that tests should be **Fast**). So, we want to focus on boundary cases, places *near* the valid cases. In this case, we can think of a couple of candidates:

- 'H': The letter immediately after the last valid note is a good candidate
- 'Z': This is a "belt and suspenders" test. It probably isn't necessary, but it is a boundary on the alphabet AND a random additional letter
- '0': This is a representative number, and also a boundary
- 'a': This would be a valid note if it was uppercase, by testing this, we would be adding a firm requirement that only uppercase letters are accepted (test as specification)

Write a test called "validSong: rejects invalid characters". Note that each one of those cases needs to be in a separate assertion, otherwise the first bad character would mask all of the others.

In addition to these, we should test 'AB', which would codify the requirement that notes are separated by spaces. Write a fourth test called "validSong: notes must be separated by spaces". Note that this time we must use two valid notes so we are only testing the spacing.

Run the tests with `npm test`. They should fail again, which means that we need to work a little harder.

Here is a basic implementation that checks if each note in the song is valid. Note that we are using another higher-order function in there (every).

```
const validSong = (song) => {
  const validNotes = ["A", "B", "C", "D", "E", "F", "G"];
  // helper to test individual notes
  const validNote = (note)=>{
    return validNotes.includes(note);
  }

  // convert the string to an array of notes
  const songList = song.split(" ");

  // every returns true if the passed in function returns true for every value
```

Copy

```
  const valid = songList.every(validNote);

  return valid;
}
```

Iterate again

Now we need to add in the sharps and flats. Because of the earlier tests, we only need to test if sharps and flats work with valid notes. We also don't need to test all possible combinations.

Tests we *should* perform:

- Are "A#" and "Ab" valid (we will assume that if it works for A, it will work for the others)
- Are the special cases invalid ("B#", "Cb", "E#", and "Fb")

Write two more tests: "validSong: sharps and flats are accepted" and "validSong: special cases are rejected". Remember that we can group multiple inputs that we expect to pass together as any single failure will cause the assertion to fail (though we lose some feedback back where the error might have happened), but we can't group inputs we expect to return `false` as they will mask each other (i.e., the first `false` means the others are never checked).

Run `npm test` again to make sure it fails (though the special character will not yet).

Let's update our function to satisfy these tests.

Copy

```
const validSong = (song) => {
  const validNotes = ["A", "B", "C", "D", "E", "F", "G"];
  const badNotes = ["B#", "Cb", "E#", "Fb"];
  // helper to test individual notes
  const validNote = (note)=>{
    // make sure the first character is valid
    let valid = validNotes.includes(note[0]);

    // check the second character
    if (note.length === 2){
      // make sure the second character is  "#" or "b"
      valid = valid && (note[1] === "#" || note[1] === "b");

      // make sure it isn't a bad note
      valid = valid && ! badNotes.includes(note);
    }

    return valid;
  }

  // convert the string to an array of notes
  const songList = song.split(" ");

  // every returns true if the passed in function returns true for every value
  const valid = songList.every(validNote);
```

```
  return valid;
}
```

Run `npm test` again. Everything should pass.

Fixing bugs

One of our users is using the function, forgets to put spaces in, and is surprised when our function accepts it.

```
> validSong('A#BB')
true
```

You can (and should) try this too.

Add a test that tests this case and demonstrate that it does fail.

Then fix the function (you should be able to do this with a simple `else if` in the note validator).

Check the coverage

You can evaluate how comprehensive your test suites are with Jest's built-in coverage reports. Run `npx jest --coverage`. Your one function should be 100% covered! But as I discussed in lecture, coverage alone is limited measure of test quality. A high quality test suite will have high coverage but a high coverage test suite does not guarantee high quality.

Running a linter

[Linters][lint] help us identify "programming errors, bugs, stylistic errors, and suspicious constructs". For this practical we will use ESLint and the AirBnB ESLint configuration. You and I may not agree with all of AirBnB's (opinionated) settings, but they provide a good starting point. It is OK for us to deviate from their recommendations, but we should do so as a considered decision.

Install ESLint and the AirBnB configuration as a development dependency by running the following command in the root directory of your package (the directory that contains the `package.json` file):

```
npm install -D eslint eslint-config-airbnb-base eslint-plugin-import
```

To configure ESLint you need to create a new file named `.eslintrc.json` in the root directory of your package with the following contents. Note that the file name is important as ESLint will look for a file with that exact name.

Copy

```
{
  "extends": "airbnb-base",
  "env": {
    "node": true,
    "jest": true
  },
  "rules": {
    "quotes": ["warn", "double"]
```

```
    }
}
```

This configuration specifies that you want to use the AirBnB base configuration and that the Node.js and Jest global variables should be predefined.

I've also added a `rules` section to show how we can customize and build on rule sets like the AirBnB base. We are adding a rule here that says all strings should use double quotes, and the linter should issue a warning if they do not.

To prevent ESLint from trying to analyze the files you created as part of the coverage analysis you will want to also create a file named `.eslintignore` file with the following list of directories (or files) to be ignored. As with `.eslintrc.json`, this file should be created in the root directory of your package.

```
# testing
/coverage
```

Copy

Just as we did for testing, you want to add a script entry point to run the linter. Add

```
"lint" : "eslint ."
```

to the scripts section of your `package.json` file, i.e. it should now look like:

```
"scripts": {
  "test": "jest",
  "lint": "eslint ."
},
```

Running the linter

Run the linter with `npm run lint` (which is equivalent to `npx eslint .`). I suspect you may have some errors! ESLint can fix many of the formatting errors automatically by running `npm run lint -- --fix`. Other errors will require you to manually refactor your code. To learn more about a particular error, Google the rule name, e.g. `no-console`. As pedantic as the formatting requirements may seem, enforcing a consistent style is very helpful in a team context. It is much easier to read your teammate's code when everyone uses the same style.

You will probably be able to eliminate all of the errors using `--fix`. However, sometimes there will be linting errors that we can't eliminate. For example, most linting rule require `console.log()` calls be removed from production code, but you may be writing something that requires it. If, after *very* careful consideration you decide that you the rule shouldn't apply, there are ways to disable certain rules. You can do so in a variety of ways, including globally (in `.eslintrc.json`), for an entire file (with a comment at the top) and for a single line (with an inline comment). For example to turn off the warnings about the console add the following comment to the top of your `index.test.js` file.

```
/* eslint-disable no-console */
```

Alternately you can add `// eslint-disable-line` to the offending line to disable ESLint on that line.

Use this power as little as possible and in the most targeted way possible. On most assignments a requirement is that your code passes the linter without errors or warnings. That shouldn't be true just because you disabled the linter...

Install Prettier

Individuality is great, but not always when you are working on a team. The linter goes a long way towards making sure your team is writing consist code, a formatter like Prettier can help as well. A formatter isn't looking at the content of your code like ESlint, just the visual formatting (primarily how white spaces and line breaks are used). Under the hood it tokenizes your code and then outputs a new version with its formatting.

Install Prettier with:

```
npm install -D prettier eslint-config-prettier
```

You can run Prettier on your file with

```
npx prettier index.js
```

Note that this doesn't actually update your file, it just dumps a formatted version to the terminal. Try it out. Add some extra new lines, change the spacing of your expression. Just make your code ugly (without changing any of the content). Then run Prettier again. You should still see a nicely formatted output.

Automate linting and formatting

The linter and the formatter are great tools, but they only work if you use them. VSCode can be configured to run both in real time and give you feedback as you work. It probably is already showing you the lint errors live. I don't recommend turning on its support for Prettier however, I found it interrupted my workflow too much (I like using white space to separate piece of code while I'm working, and Prettier wants to collapse them).

Instead, we are going to set them up so that they are run when you commit your code to git (that way every commit is always nicely formatted).

We will install a new tool called husky.

```
npm install -D husky lint-staged
```

Husky will add "hooks" to the git process, and then it will call the other package you installed (lint-staged) when the time comes.

We are going to add two more things to your `package.json` file. In the scripts section, add `"prepare": "husky install"`. The `prepare` script is a special one that will be run when you do `npm install`. This allows husky to set itself up in a new deployment. Run the script now with `npm run prepare`. You should see something like:

```
practical02-test> npm run prepare

> practical02@1.0.0 prepare
> husky install
```

```
husky - Git hooks installed
```

Then add the following to `package.json`

```
"lint-staged": {
  "*.js": "eslint --cache --fix",
  "*": "prettier --ignore-unknown --write"
}
```

It should go at the top level of the object. This says that we would like to run every JavaScript file through the linter, and every file through Prettier.

The last piece here is to actually add the hook to husky. Run the following:

```
npx husky add .husky/pre-commit "npx lint-staged"
```

That tells husky that we would like the `lint-staged` command that we just configured to be run before anything is committed to git. Note that if the linter finds an error and fails, the commit will fail and you will have to fix it before you can proceed.

Go ahead and commit your work to git now (recall from last time it is a two step process, you have to stage your changes them commit them). You should see the `lint-staged` tasks run at the start of the commit.

Finishing up

Commit any changes you may have made since the commit at the end of the last section and then push your changes to GitHub.

You should then submit your repository to Gradescope.

Grading

- Create a npm package
- Implement `validSong`
- Implement tests with 100% coverage
- Pass all tests
- Pass all ESLint checks

Recall that the Practical exercises are evaluated as "Satisfactory/Not yet satisfactory". Your submission will need to implement all of the required functionality (i.e., pass all the tests) to be Satisfactory (2 points). Note that we don't have a way to test your tests, so this practical exercises incorporates manual review.
