Exercise 2 - TDD in a framework

Description

We'll continue working with Vue/VueX and write actions tests. This time, we will have an application that should be able to fetch TV series data and present it, based on a search query.

The major difference from before is that we have an external dependency, which calls a third-party service, grabs the data in an AJAX call and returns the result as an array.

In this case, we'll need to mock the service to be able to control the data that we use in our tests. The service returns an array on each call, looking like this:

```
[
  {
    "name": "name of show",
    "image": "url to image",
    "summary": "summary of show",
    "rating": {
      "percentage": 50
    }
  },
  ...
]
```

Steps

- Enter the `exercise-3` folder and run `npm i`
- Start the application by running `npm start`.
- Point your browser to `http://localhost:9000/`. As you can see, there's a search bar and a button. Clicking the button won't do anything right now, except write a debug message to the console.
- To get acquainted with the code, open `src/App.vue`. The button is wired up to an action called `onSearch`, which is dispatched in the function `search()` in `App.vue`. This application only uses this single dispatch, and as before, you don't have to wire it up yourself.
- Open `src/services/showSearch.js` and take a look. It fetches TV series information when passed a search string, re-maps it into something more useful to our application, and returns it. This is the service that we will be mocking in tests.

- Open the file `src/store/modules/shows.js`. This is where the logic we *want to test* will live, in the actual action. Go ahead and remove the `console.log` line in preparation for the code we'll write later.
- Finally, let's write some tests. Let's start with a test that *makes sure we call our service with the search string*. To do that, we need to dispatch an action, plus setup a mock. Start by opening the file `test/all-tests.js`.
- Add a test, such as:

```
it('should call the TV series service and pass along the search string', async () => {
    const sut = createStore();
    const allArgs = [];

    mock({
        query(searchString) {
            allArgs.push(searchString);
            return Promise.resolve([]);
        }
    });

    await sut.dispatch('onSearch', 'stranger things');
    assert.strictEqual(allArgs.length, 1);
    assert.strictEqual(allArgs[0], 'stranger things');
});
```

- You will also need the following line at the top of your test file:

```
import { mock } from '../src/services/showSearch';
```

- Run the test with `npm t` as usual. It should fail right now.
- Make the test pass, by adding the necessary code in `shows.js`. Note that the `getService` function is already included and callable right away (which is the function that return the service, or in our case, the mock, that you can call `query` on). To do this in *true TDD style*, don't do more than necessary. You can get this test to pass without even modifying the state, by just simply calling the service with the right argument.
- When your test pass, make another test, which returns something different than an empty list (instead of the empty array in `Promise.resolve([])`). This test should assert that `sut.state.shows.searchResult` contains the data returned from the service. Check the top of this document to remind yourself what the format looks like. Add a fake entry or two, and assert that the store contains them after the dispatch call. Don't forget that you will need to add an `async` specifier to the function, and await the call to `query`. Then call `commit` with the list retrieved from the server as payload - there's a mutation called `updateSearch` prepared for you.

- If everything passes, your app should work (mostly) as expected. Give it a try by typing a search string in the browser.

Stretch task

There are two CSS classes (`good` and `bad`) in `App.vue`. Augment the array of items in your `mutation` before adding them to the store, to include a `rating.class` (alongside the `percentage` field). This should be `good` if rating is >= 50%, and otherwise `bad`.

Don't forget to write one failing test before writing the implementation for each case!