test-asynchronous-code-jest

Asynchronicity is a fundamental concept of the web today. Due to the single-threaded nature of the JavaScript event loop, executing code in a non-blocking way is critical to building efficient and fast programs. In this guide, we will explore the different ways Jest give us to test asynchronous code properly.

false positives examples :

- A pregnancy test is positive, when in fact you aren't pregnant.
- A cancer screening test comes back positive, but you don't have the disease.

False Positives on Tests

```
test("this shouldn't pass", () => {
  setTimeout(() => {
    // this should fail:
    expect(false).toBe(true);
  });
});
```

The above test is a false positive. The test will pass but the assertion ***should*** make it fail. Jest will run the test function and, since the test function simply starts a timer and then ends, Jest will assume the test passed. The test completes early, and the expectation will run sometime in the future when the test has already been marked as passed.

We need to inform Jest that it should wait until the assertions run. How we do this depends on which asynchronous patterns the code we want to test uses.

Asynchronous Patterns

There are several patterns for handling async operations in JavaScript; the most used ones are:

- Callbacks
- Promises
- Async/Await

For callback-based code, Jest provides a `done` callback as the argument of the test function. We should invoke this function after we are done asserting.

For example, let's say we have an asynchronous `add` function which waits for a half-second to produce a result:

```javascript
function addAsync(a, b, callback) {
  setTimeout(() => {
    const result = a + b;
    callback(result);
  }, 500)
}
```

We can't test this as a function like this:

We can test the above function like this:

```
1  test('add numbers async', done => {
2    addAsync(10, 5, result => {
3      expect(result).toBe(15);
4      done();
5    })
6  })
```

Since the test function defines the `done` parameter, Jest will wait for expectations until `done()` is invoked. Jest, by default, will wait for up to five seconds for a test to complete. Otherwise, the test will fail.

Testing Promises

The simplest way to let Jest know that we are dealing with asynchronous code is to return the Promise object from the test function. You can, for example, evaluate the expectations in the `then` callback:

```javascript
//...
test('properly test a Promise', () => {
  return somePromise.then(value => {
    expect(value).toBeTrue();
  })
})
```

Jest also provides the `resolves` / `rejects` matchers to verify the value of a promise. They are convenient syntax sugar that allows us to write code like this:

```javascript
test('should resolve to some value', () => {
  const p = Promise.resolve('some value');
  return expect(p).resolves.toBe('some value');
});

test('should reject to error', () => {
  const p = Promise.reject('error');
  return expect(p).rejects.toBe('error');
});
```

Note that these matchers also return a Promise object; that's why we must return the assertion. If we don't return it, we will have false positives again.

Testing With Async / Await

As we saw in the previous section, Jest will know that we are dealing with asynchronous code if we return a Promise object form the test function. If we declare the test function as `async`, it will implicitly make the function to return a Promise. We can also use the `await` keyword to resolve Promise values and then assert them as if they were synchronous.

For example, we can wait for the resolved Promise value and assert it, like so:

javascript

```
test('shows how async / await works', async () => {
  const value = await Promise.resolve(true);
  expect(value).toBe(true);
});
```

/ We

This approach is very convenient. It lets you run the expectations just as if the values were synchronous. In the end, since we are waiting for the asynchronous values, the Promise that the test function returns will make Jest aware of the need to wait. Also, if you miss the await keyword, the test will fail because it is expecting some value, not a Promise.

Handling Nested Promises Using Async/Await in React

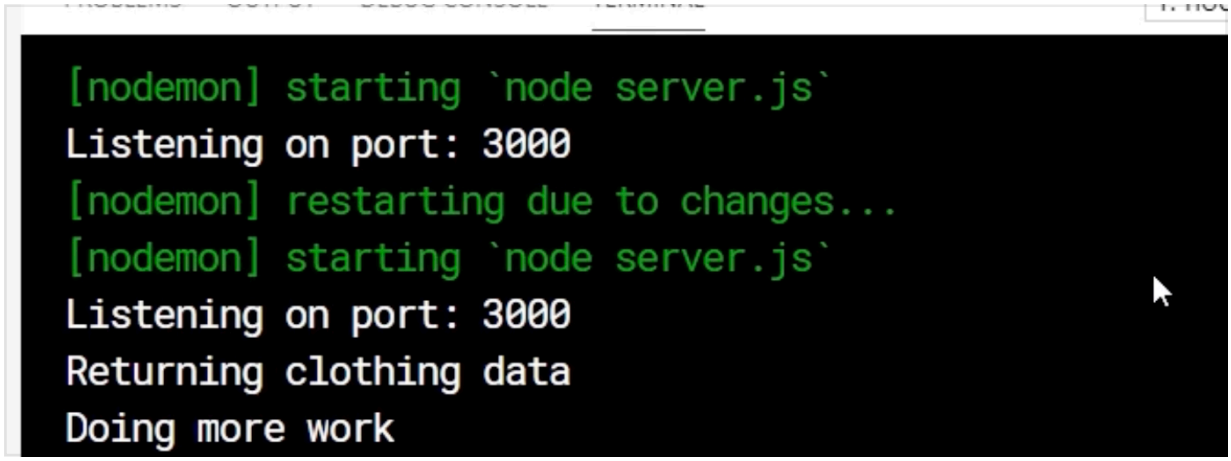
have to get chips after we get fish...
2getFishAndChips = async () => {
3 const fish = await fetch(this.fishApiUrl).then(response => response.json());
4
5 const fishIds = fish.map(fish => fish.id),

```
6    chipReqOpts = { method: 'POST', body:
JSON.stringify({ fishIds }) };
7
8    const chips = await fetch(this.chipsApiUrl,
chipReqOpts).then(response => response.json());
9}
```

— Sync code : fs.readFileSync

```
/* GET all clothing */
router.route('/')
  .get(function(req, res) {

    let rawData = fs.readFileSync(datafile, 'utf8');
    let clothingData = JSON.parse(rawData);
    console.log('Returning clothing data');
    res.send(clothingData);
    console.log('Doing more work');
  });

module.exports = router;
```

Output:



Convert same code to Async using readFie is async

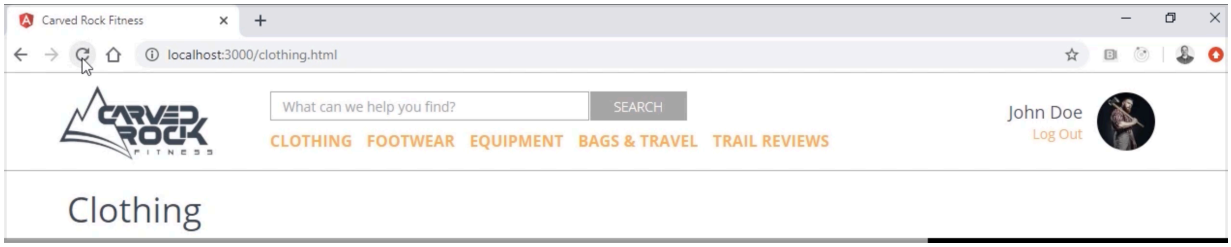
```
.get(function(req, res) {

  fs.readFile(datafile, 'utf8', (err, data) => {
    if (err) {
      console.log(err);
    }
    else {
      let clothingData = JSON.parse(data);
      console.log('Returning clothing data');
      res.send(clothingData);
    }
  });
  console.log('Doing more work');
});
```

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Listening on port: 3000
Doing more work
Returning clothing data
```

Accepting callbacks as parameters

```
/* GET all clothing */
router.route('/')
  .get(function(req, res) {

    let clothingData = getClothingData();

    console.log('Returning clothing data');
    res.send(clothingData);
    console.log('Doing more work');
  });

function getClothingData() {
  fs.readFile(datafile, 'utf8', (err, data) => {
    if (err) {
      console.log(err);
    }
    else {
      let clothingData = JSON.parse(data);
      return clothingData;
    }
  });
}
```

Output:



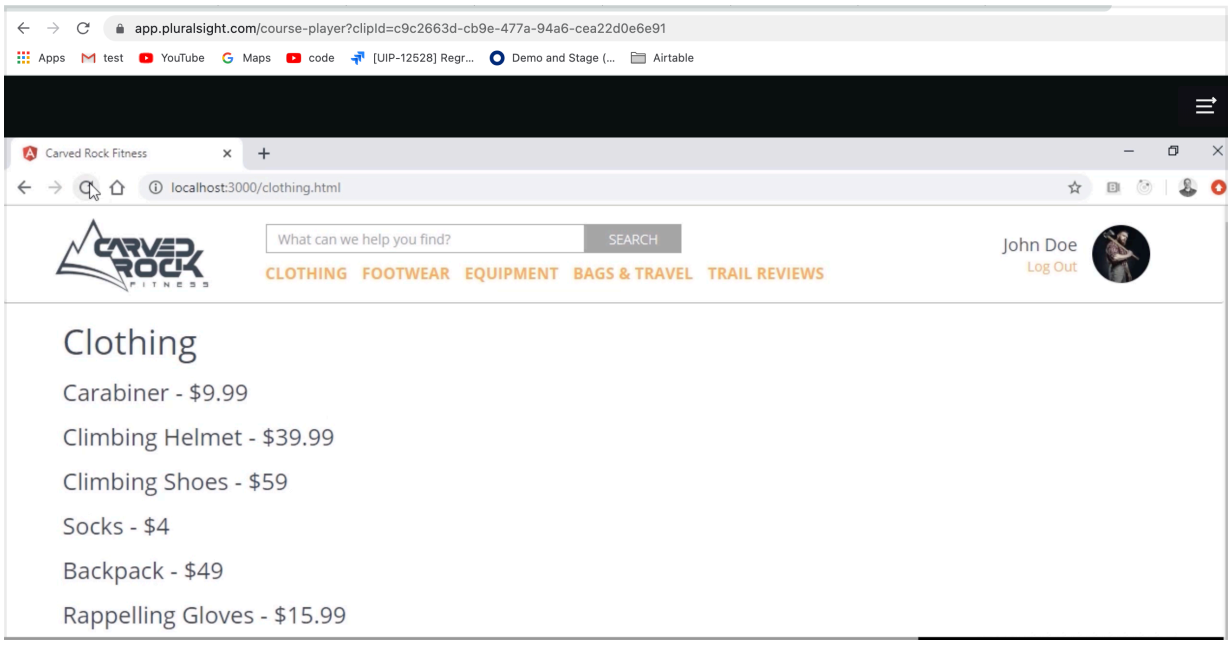
Not Data present, because clothingData is undefined and still in progress and as its async call

Fix the above problem

```
/* GET all clothing */
router.route('/')
  .get(function(req, res) {

    getClothingData((err, data) => {
      if (err) {
        console.log(err);
      }
      else {
        console.log('Returning clothing data');
        res.send(data);
      }
    });

    console.log('Doing more work');
  });

function getClothingData(callback) {
  fs.readFile(datafile, 'utf8', (err, data) => {
    if (err) {
      callback(err, null);
    }
    else {
```

```
        let clothingData = JSON.parse(data);
        callback(null, clothingData);
      }
    });
}
```

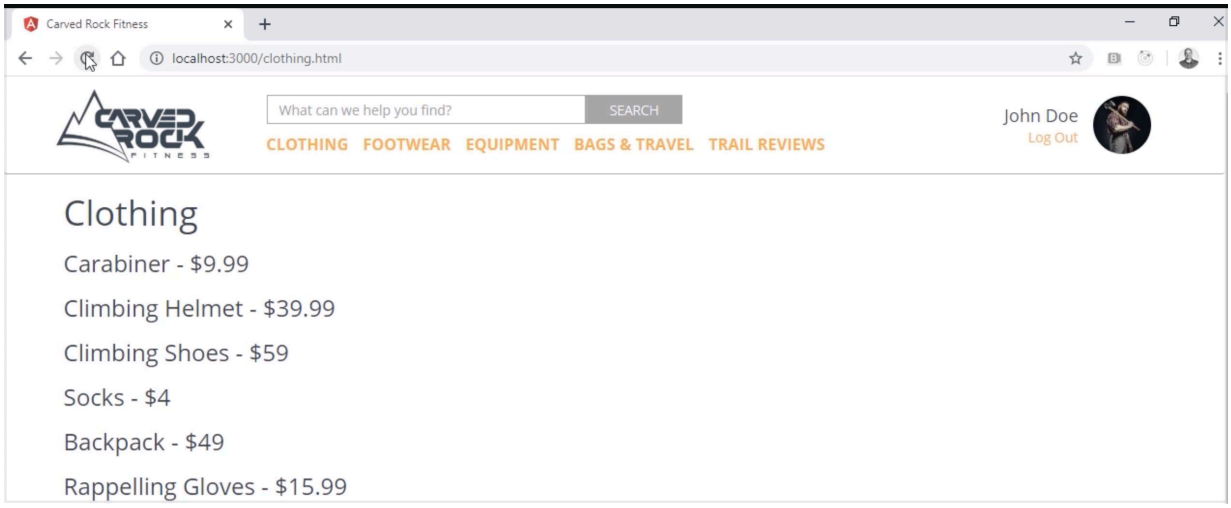
Output — now Browser showing up

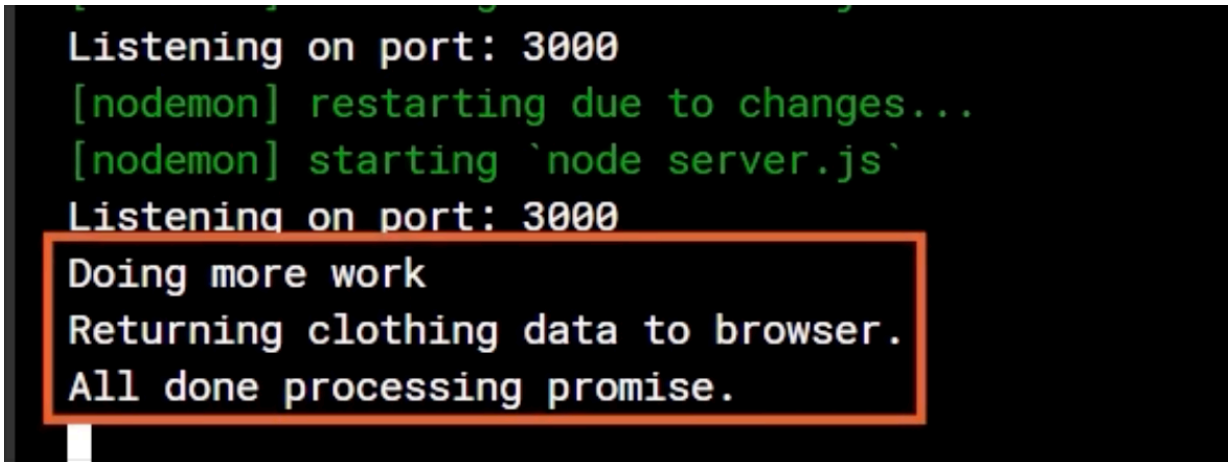


Handle same code with Promise

```
const datafile = 'server/data/clothing.json';
const router = express.Router();

/* GET all clothing */
router.route('/')
  .get(function(req, res) {

    getClothingData()
      .then(data => {
        console.log('Returning clothing data to browser.');
        res.send(data);
      })
      .catch(error => res.status(500).send(error))
      .finally(() => console.log('All done processing promise.'));

    console.log('Doing more work');
  });
```

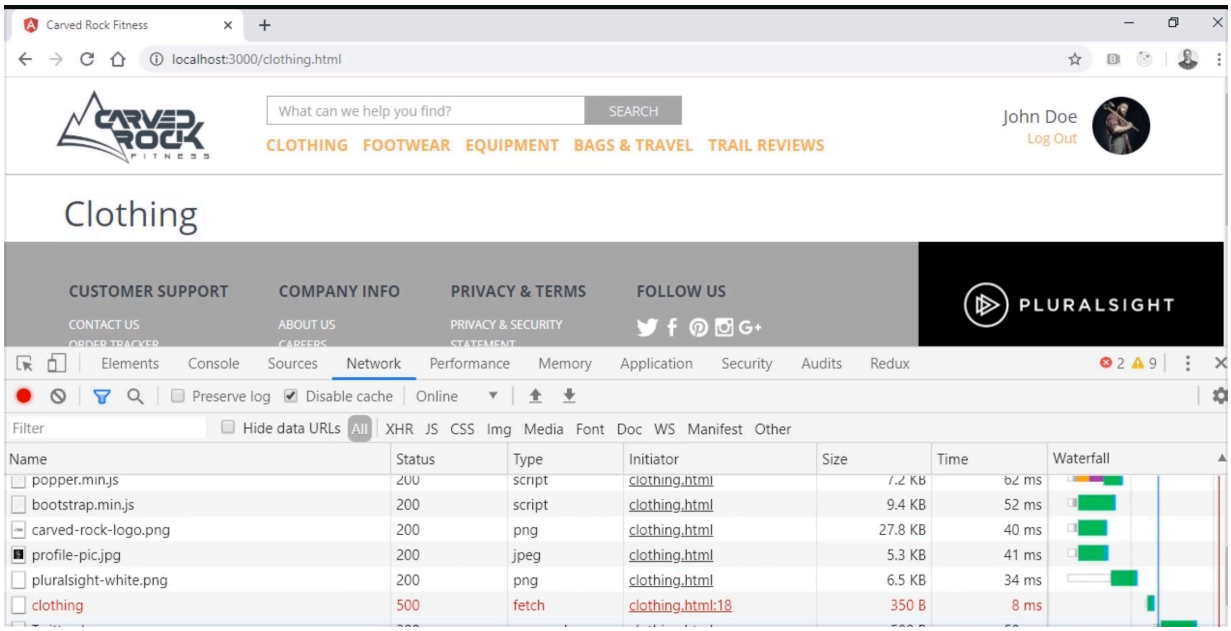
```
function getClothingData() {

  return new Promise((resolve, reject) => {
    fs.readFile(datafile, 'utf8', (err, data) => {
      if (err) {
        reject(err);
      }
      else {
        let clothingData = JSON.parse(data);
        resolve(clothingData);
      }
    });
```

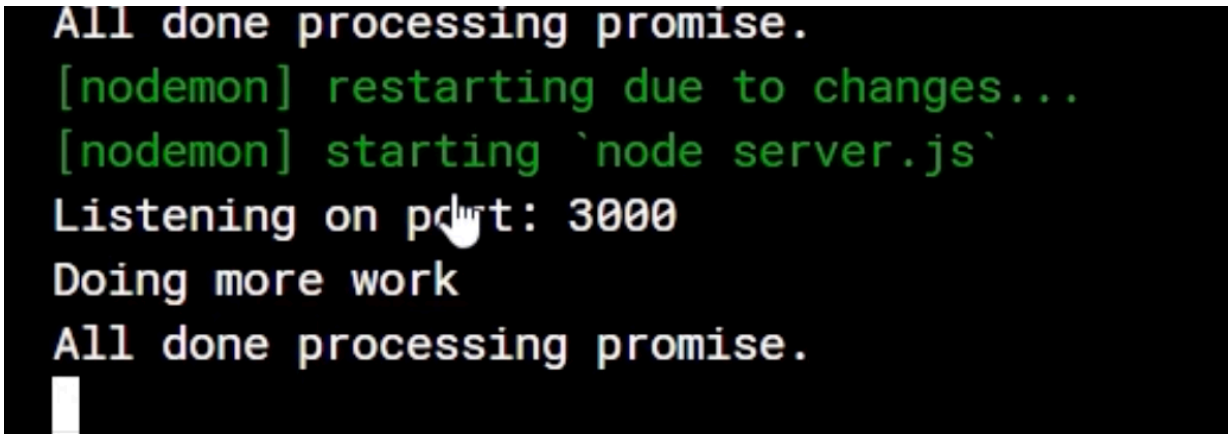




Replicate Promise Reject error by changing file name and Doesn't exist

```
const datafile = 'server/data/bad_name.json';
```





What is Generator Function

Generator Functions

Functions that can be paused and resumed
- State of the function is stored while paused

Return generators
- Implement the iterator protocol

Lazy execution
- Values computed on demand

Handle same with Async / Await

async/await

Built with promises and generators

Reads more like synchronous code

Data returned from async functions is automatically wrapped in a promise

Using the await keyword will automatically extract data from a promise

```
async function getClothingData() {

  let rawData = await fsPromises.readFile(datafile, 'utf8');
  let clothingData = JSON.parse(rawData);

  console.log(clothingData);

  return clothingData;

  // return new Promise((resolve, reject) => {
  //   fs.readFile(datafile, 'utf8', (err, data) => {
  //     if (err) {
  //       reject(err);
  //     }
  //     else {
  //       let clothingData = JSON.parse(data);
  //       resolve(clothingData);
  //     }
  //   });
  // });

}
```

```
/* GET all clothing */
router.route('/')
  .get(async function(req, res) {

    try {
      let data = await getClothingData();
      console.log('Returning async data.');
      res.send(data);
    }
    catch (error) {
      res.status(500).send(error);
    }

    // getClothingData()
    //   .then(data => {
    //     console.log('Returning clothing data to browser.');
    //     res.send(data);
    //   })
    //   .catch(error => res.status(500).send(error))
    //   .finally(() => console.log('All done processing promise.'));

    console.log('Doing more work');
```

Found all the code :

```
QT71QQYQ2C:~ rsreenadhu$ cd Downloads/nodejs-async-patterns.zip
```

javascript-promises-async-programming.zip

#pluralsight