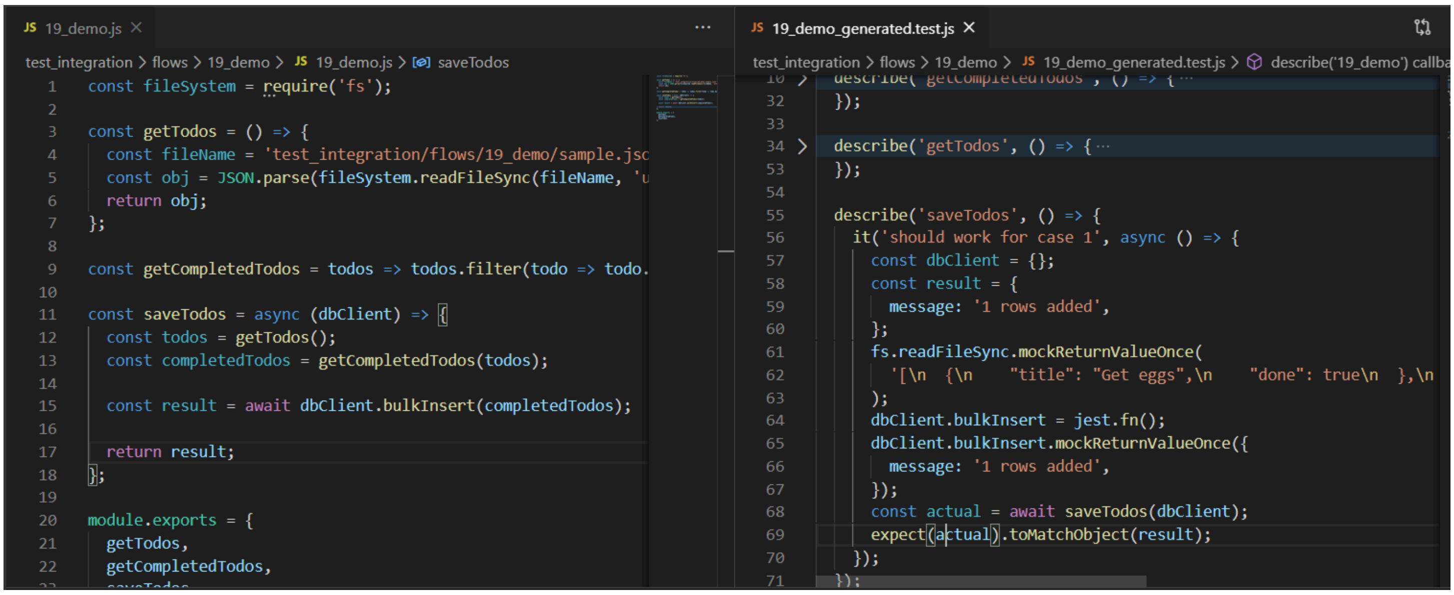
Unit Test Recorder - Automatically generate unit tests as you use your application

25 Jun 2020

Introduction



19_demo.js

test_integration > flows > 19_demo > 19_demo.js > saveTodos

```javascript
const fileSystem = require('fs');

const getTodos = () => {
  const fileName = 'test_integration/flows/19_demo/sample.jso
  const obj = JSON.parse(fileSystem.readFileSync(fileName, 'u
  return obj;
};

const getCompletedTodos = todos => todos.filter(todo => todo.

const saveTodos = async (dbClient) => {
  const todos = getTodos();
  const completedTodos = getCompletedTodos(todos);

  const result = await dbClient.bulkInsert(completedTodos);

  return result;
};

module.exports = {
  getTodos,
  getCompletedTodos,
```

19_demo_generated.test.js

test_integration > flows > 19_demo > 19_demo_generated.test.js > describe('19_demo') callba

```javascript
describe('getCompletedTodos', () => {···
});

describe('getTodos', () => {···
});

describe('saveTodos', () => {
  it('should work for case 1', async () => {
    const dbClient = {};
    const result = {
      message: '1 rows added',
    };
    fs.readFileSync.mockReturnValueOnce(
      '[\n  {\n    "title": "Get eggs",\n    "done": true\n  },\n
    );
    dbClient.bulkInsert = jest.fn();
    dbClient.bulkInsert.mockReturnValueOnce({
      message: '1 rows added',
    });
    const actual = await saveTodos(dbClient);
    expect(actual).toMatchObject(result);
  });
});
```

Input code on left and generated code on right.

This is a write up for the npm package unit-test-recorder or UTR for short.

The package is a CLI tool, which enables the user to record the activities flowing through every function in the nodejs application. This can then be turned into working unit test cases by string interpolation.

This tool is meant to be used for regression testing using snapshots.

The generated snapshots are human readable and editable code. The users are encouraged to take it forward and make changes to this code directly.

Methodology

In testing, we pass some arguments to the function under test and expect some result in return. If we are confident that the function in question is stable, then we can record the parameters and result as ground truth.

In order to record the parameters and results, we need to make some changes to the code itself. This process is called instrumentation. Instrumenters are used in many places including code coverage runners like istanbuljs and monitoring services like new relic.

UTR uses babeljs for injecting the recorder in the code.

Recording functions

Lets take this function as an example

```
const foo = (a, b) => a + b;
```

UTR uses babel to explicitly instrument it to

```
const foo = (...p) => recorderWrapper(
  { name: 'foo', fileName: 'bar' },
  (a, b) => a + b,
  ...p,
);
```

The `recorderWrapper` function is provided by UTR. The function below is a simplified version of its implementation.

```
const recorderWrapper = (meta, fn, ...p) => {
  const result = fn(...p);
  Recorder.record({ meta, result, params: p });
  return result;
};
```

The meta property stores additional information regarding the nature of the function. e.g. The file name where the function was located, whether it returns a promise, whether it is the default export or named export and alike.

This function records all calls to this function in a state. Here is a simplified representation of the state.

```
{
  "fileName":{
    "functionName":{
      "captures":[
        { "params": [1, 2], "result": 3 },
        { "params": [2, 3], "result": 5 },
      ]
    }
  }
}
```

Now using string interpolation, we can generate test cases.

```
describe('fileName', () => {
  describe('functionName', () => {
    it('test 1', () => {
      expect(foo(1, 2)).toEqual(3);
    });
    it('test 2', () => {
      expect(foo(2, 3)).toEqual(5);
    });
  });
});
```

Recording mocks

For recording mocks, we need to record all invocations of the imported function and correlate it to the function we are currently recording.

```
const fs = require('fs');
const foo = (fileName) => fs.readFileSync(fileName).toString().toUpperCase();
const bar = (fileName) => fs.readFileSync(fileName).toString().toLowerCase();
const baz = (f1, f2) => foo(f1) + bar(f2);
```

The problem is that, if we instrument the mocked function naively, we would have no way to know which function invoked the mocked function.

```
fs.wrappedReadFileSync = (...p) => recorderWrapper(
  { name: 'fs.readFileSync' },
  fs.readFileSync, // There is no way to know if foo invoked the function or bar or baz
  ...p,
);
const foo = (...p) => recorderWrapper(
  { name: 'foo' },
  (fileName) => fs.wrappedReadFileSync(fileName).toString().toUpperCase(),
  ...p,
);
const bar = (...p) => recorderWrapper(
  { name: 'bar' },
  (fileName) => fs.wrappedReadFileSync(fileName).toString().toUpperCase(),
  ...p,
);
const baz = (...p) => recorderWrapper(
  { name: 'baz' },
  (f1, f2) => foo(f1) + bar(f2),
  ...p,
);
```

Therefore the `recorderWrapper` for the `fs.readFileSync` must be aware of the JavaScript call stack or the continuation. Continuation local storage is a popular JavaScript library built for this purpose. However in nodejs 13 and onward, native support exists in the form of AsyncLocalStorage.

To illustrate we now use two separate wrappers `mockRecorderWrapper` and `functionRecorderWrapper`.

```
const functionRecorderWrapper = (meta, fn, ...p) => {
  // Set the meta so that all invoked functions have access to it
  cls.set('meta', meta);
  return recorderWrapper(meta, fn, ...p);
};
const mockRecorderWrapper = (mockMeta, fn, ...p) => {
  // Get the meta of the function that invoked this mock
  const invokerMeta = cls.get('meta');
  // Use invoker meta to make sure the recordings are correlated correctly
  return recorderWrapper({ ...mockMeta, ...invokerMeta }, fn, ...p);
};
```

Now, if the mocked function is invoked multiple times, we can save the sequence in an array

```
{
  "foo": [
    {"moduleName": "fs", "functionName": "readFileSync", "params": [ "file1" ], "result": "file1_c
  ],
  "bar": [
    {"moduleName": "fs", "functionName": "readFileSync", "params": [ "file1" ], "result": "file1_c
  ],
  "baz": [
    {"moduleName": "fs", "functionName": "readFileSync", "params": [ "file1" ], "result": "file1_c
    {"moduleName": "fs", "functionName": "readFileSync", "params": [ "file2" ], "result": "file2_c
  ]
}
```

The generated tests look like so

```
const fs = require('fs');
jest.mock('fs');
describe('fileName', () => {
  describe('baz', () => {
    it('test 1', () => {
      fs.readFileSync.mockImplementationOnce('file1_contents');
      fs.readFileSync.mockImplementationOnce('file2_contents');
      // baz is invoking foo and bar. They inturn are consuming the mocks
      expect(baz('file1', 'file2')).toEqual('FILE1_CONTENTSfile2_contents');
    });
  });
});
```

Dependency injections

Another common programming pattern in JavaScript involves using dependency injections. Often database clients are passed to functions like the example below.

```
const foo = async (dbClient) => {
  const rows = await dbClient.query('SELECT COUNT(*) as usercount FROM users;');
  return rows[0].usercount;
};
```

In order to generate tests for the function `foo`, we need to record the `query` function of the `dbClient` argument. However, if we modify the `dbClient.query` function itself, it can lead to unintended side effects. Instead we create a new function at run time and use babel to modify the source code to make sure the new created function will be used.

```
const foo = async (dbClient) => {
  const rows = await dbClient.wrappedQuery('SELECT COUNT(*) as usercount FROM users;');
  return rows[0].usercount;
};
```

The code below is a simplified version of the actual implementation

```
// Executed before the function is executed
const preExecutionHook = (wrappingMeta) => {
  // wrappingMeta looks similar to { objectName: 'dbClient', fnName: 'query' };
  const { fnName } = wrappingMeta;
  const newFnName = generateNewFnName(wrappingMeta); // "wrappedQuery" in this case
  // dbClient.wrappedQuery created at runtime
  dbClient[newFnName] = (...p) => {
    const invokerMeta = cls.get('meta');
    const fn = dbClient[fnName];
    return recorderWrapper({ ...wrappingMeta, ...invokerMeta }, fn, ...p);
  };
};
```

In the generated tests, we mock the methods of the injected object like so

```
describe('fileName', () => {
  describe('foo', () => {
    it('test 1', () => {
      const dbClient = {};
      dbClient.query.mockImplementationOnce([{ usercount: 20 }]);
      expect(foo(dbClient)).toEqual(20);
    });
  });
});
```

Injection promotions

To make the test generation process data efficient, we use injection promotions. In the example below, even if the function bar is never invoked separately, we can piggyback on the invocation of foo to generate tests for both foo and bar.

```
const bar = (client, status) => client.query('SELECT COUNT(*) FROM users WHERE status=$1', status)
const foo = async (dbClient) => {
  const offlineUsers = await bar(dbClient, 'offline');
  const activeUsers = await dbClient.query('SELECT COUNT(*) FROM users WHERE status=$1', 'active')
  const awayUsers = await bar(dbClient, 'away');
  return { offlineUsers, activeUsers, awayUsers };
};
```

Although testing foo also covers bar, the separate tests for bar act as scaffolding for any future modifications to the function bar. Hence, the design decision.

The generated tests look like so

```
describe('fileName', () => {
  describe('bar', () => {
    it('test 1', () => {
      const dbClient = {};
      dbClient.query.mockImplementationOnce(10);
      expect(bar(dbClient, 'offline')).toEqual(10);
    });
    it('test 2', () => {
      const dbClient = {};
      dbClient.query.mockImplementationOnce(30);
      expect(bar(dbClient, 'away')).toEqual(30);
    });
  });
  describe('foo', () => {
    it('test 1', () => {
      const dbClient = {};
      dbClient.query.mockImplementationOnce(10); // Promoted from bar
      dbClient.query.mockImplementationOnce(20);
      dbClient.query.mockImplementationOnce(30); // Promoted from bar
      const expected = { offlineUsers: 10, activeUsers: 20, awayUsers: 30 };
      expect(foo(dbClient)).toEqual(expected);
    });
  });
});
```

Promotions are handled internally by maintaining an implicit stack. When a function's scope ends, all the recorded dependency injections are passed on to its parent.

We use uuidv4 to correctly identify injected functions for promotions as variable names need not be consistent between parent and child.

1st invocation to dbClient.query

```
fooCaptures = {
  captures: [], // foo has not called dbClient.query yet
};
barCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
  ],
};
```

```
// When bar finishes, the captures are promoted to its parent
fooCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
  ],
};
```

2nd invocation to dbClient.query

```
// When foo invokes dbClient.query directly, it gets added to the captures array
fooCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'active',
      ],
      result: 20,
    },
  ],
};
```

3rd invocation to dbClient.query

```
// Foo stack so far
fooCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'active',
      ],
      result: 20,
    },
  ],
};
// When bar is invoked, the injections are captured for bar
barCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'away',
      ],
      result: 30,
    },
  ],
};
// Finally, when scope of bar ends, injections are promoted
fooCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'active',
      ],
      result: 20,
    },
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'away',
      ],
      result: 30,
    },
  ],
};
```

Conclusion

Other than the usual benefits of software testing, this is also a quick and inexpensive way to generate a dataset of test code, corresponding to a corpora of existing code. This can be used by researchers working on deep learning based code generation tools.

All articles

Theorem: Equivalence of -arccoth and tan Functions
21 Nov 2025

The Photonic Transformer Hypothesis: Rethinking Photosynthesis and Methane Consumption in Plants
19 Nov 2025

The Scalable Quasi-Perpetual Photonic Machine
17 Nov 2025

How to Make a Ball Orbit Itself
08 Nov 2025

The Boron-Nitrogen Catastrophe: A Critical Gap in Beta Decay Verification
28 Oct 2025

X-Ray Data Pipeline: Ultra-High-Speed Communication Through Limestone Tubes
26 Oct 2025

The Universal Approximation Theorem Is Right. You're Using It Wrong.
19 Oct 2025

The Power Law Illusion: A Measurement Artifact Hypothesis
11 Oct 2025

Fractional Gamma Function via Fractional Derivatives
18 Sep 2025

On the cruel irony of the P-NP problem
18 Dec 2023

The Journey from India to Germany: A Guide for IT Professionals
20 Jun 2023

How does AutoGPT work under the hood?

03 May 2023

Unit Test Recorder - Automatically generate unit tests as you use your application

25 Jun 2020

How to migrate from vanilla Kubernetes to Istio service mesh?

14 Oct 2019