
## Das JavaScript-Test-Framework auf dem Prüfstand

# Test mit Jest.

Wer JavaScript nicht automatisiert testet, handelt sich schon wegen des vergleichsweise offenen Sprachstandards über kurz oder lang gravierende Probleme ein. Mit Jest gibt es nun ein neues JavaScript-Test-Framework, das aus den Fehlern der alten zu lernen sucht.

von Tam Hanna

Als sich der Autor dieser Zeilen in den frühen 2010er Jahren mehr als Hobby denn wegen ernsthafter dienstlicher Notwendigkeit mit JavaScript auseinandersetzte, lebte die Sprache im Allgemeinen in Browsern. Die Kämpfe zwischen Mozilla, Opera und dem Internet Explorer hatten dafür gesorgt, dass JavaScript Engines immer schneller wurden. Dadurch konnte man bisher am Server und auf einem nativen Client liegenden Code in Richtung der Cloud übertragen, was die Installation unnötig machte und technisch herausgeforderte Nutzer von den Supporthotlines fernhielt.

In der damaligen Zeit entstanden erste Frameworks, die damals allerdings auf die Arbeit mit webbasierten Inhalten optimiert waren. Produkte wie Node.js und andere Runtimes ließen sich damit, wenn überhaupt, nur unter großem Aufwand testen. Das Jest-Entwicklerteam möchte mit seinem Framework alles anders machen. Das beginnt damit, dass Dienste wie Babel und Node, aber auch die Frameworks Angular, React und Vue unterstützt werden. Neben der Ansage, dass sich Tests mit so gut wie keinem Konfigurationsaufwand einrichten lassen, verspricht man auch, dass die Isolation der einzelnen Testaufgaben in eigenen Threads Prüfungen von sehr großen Codebasen mit sehr hoher Performance

bewerkstelligt. Zu guter Letzt betont die unter https://jestjs.io bereitstehende Webseite des Produkts auch noch, dass eine als Snapshot bezeichnete Funktion Vergleiche von Objekten gegen Referenzvorlagen mit geringem Aufwand erledigt. Kurzum gibt es einiges, was Entwickler freuen könnte – werfen wir also einen Blick auf das Produkt.

### Eine Frage der Ausführungsumgebung

Die Dokumentation des Frameworks demonstriert die Verwendung des Produkts vorrangig unter Verwendung der Yarn-Arbeitsumgebung. Da wir in den folgenden Schritten unter Ubuntu 18.04 arbeiten, lässt sich die Installation durch den bekannten Dreikampf bestehend aus dem Addieren eines Schlüssels, dem Ausführen eines Skripts und dem Herunterladen eines Pakets bewerkstelligen:

```
t@t18:~$ curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key
                                                                    add -
t@t18:~$ echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee
                                                  /etc/apt/sources.list.d/yarn.list
t@t18:~$ sudo apt-get update && sudo apt-get install yarn
```

Yarn ist nicht nur unter Linux verfügbar. Das Entwicklerteam der Arbeitsumgebung stellt unter [1] Installationsanweisungen für andere Betriebssysteme bereit. Wer unter macOS oder Windows arbeiten möchte, muss nicht auf die Vorteile verzichten. Yarn orientiert sich von der Vorgehensweise her am von Node.js bekannten npm. Analog beginnen wir deshalb auch hier mit der Erzeugung eines neuen Projekts, das in einem leeren Unterordner unterkommen sollte:

```
tamhan@TAMHAN14:~/yarnspace$ yarn init
yarn init v1.13.0
question name (yarnspace): jesttest
question version (1.0.0):
...
success Saved package.json
Done in 7.82s.
```

Analog zum großen Vorbild gilt auch hier, dass sie diverse Anfragen im Allgemeinen durch Drücken von Enter quittieren können. Der Autor dieser Zeilen entschied sich immerhin dafür, den Ordnernamen durch einen anderen Projektname zu ersetzen. Im nächsten Schritt folgt ein Aufruf des *add*-Befehls, der das Test-Framework herunterlädt und installiert:

```
tamhan@TAMHAN14:~/yarnspace$ yarn add --dev jest
yarn add v1.17.3
...
```

An sich ist auch das keine Raketenwissenschaft. Wichtig ist nur, dass der Parameter *dev* übergeben werden muss. Er weist die Runtime darauf hin, dass die anzulegende Ressource eine *devDependency* darstellt. Das ist ein


Abb. 1: Jest startet zwar, verendet aber sofort danach

Sonderregime von Yarn, in dem das betreffende Paket nur während der Entwicklung bereitgestellt wird. Nach dem erfolgreichen Durchlaufen des Deployment-Prozesses ist die auch in npm-basierten Projekten vorhandene Datei *package.json* um einen zusätzlichen Verweis auf das Test-Framework reicher. Das lässt sich überprüfen, indem Sie es in einem Editor Ihrer Wahl öffnen. Lassen Sie das File auch gleich geöffnet, weil Sie zusätzlich die Passage aus Listing 1 in der *scripts*-Rubrik anlegen müssen.

Jest-basierte Testprozesse lassen sich in der Theorie auch aus der Kommandozeile heraus anstoßen. Die Praxis lehrt allerdings, dass Unit-Test-Prozesse vor allem dann konsequent durchgeführt werden, wenn sie für den Entwickler minimalen bis nicht vorhandenen Aufwand erzeugen. Ein schöner Weg, um Schnelligkeit sicherzustellen, ist die Einbindung von Jest in den Yarn-Workflow. Die Aktivierung des Testprozesses erfolgt dann analog zu allen anderen Kommandos des Paketmanagers; dank der Parallelisierung der Ausführung sollte die Gesamtausführungszeit zudem vergleichsweise gering sein. Damit haben wir die Arbeiten fürs erste abgeschlossen und können mit dem Test beginnen.

### Eine erste Trockenübung

Sofern Sie die Datei *package.json* nach dem Einpflegen des Skriptblocks gespeichert haben, sind wir in der Theorie zur Abarbeitung bereit. Kehren Sie auf die Kommandozeile zurück, und befehlen Sie durch Eingabe von

**Listing 1**

```
{
  ...
  "devDependencies": {
    "jest": "^24.1.0"
  },
  "scripts": {
    "test": "jest"
  }
}
```

**Listing 2**

```
function sayHello()
{
  return "Hello";
}
function add(a, b)
{
  return a+b;
}
function divide(a,b)
{
  return a/b;
}
module.exports = sayHello;
```

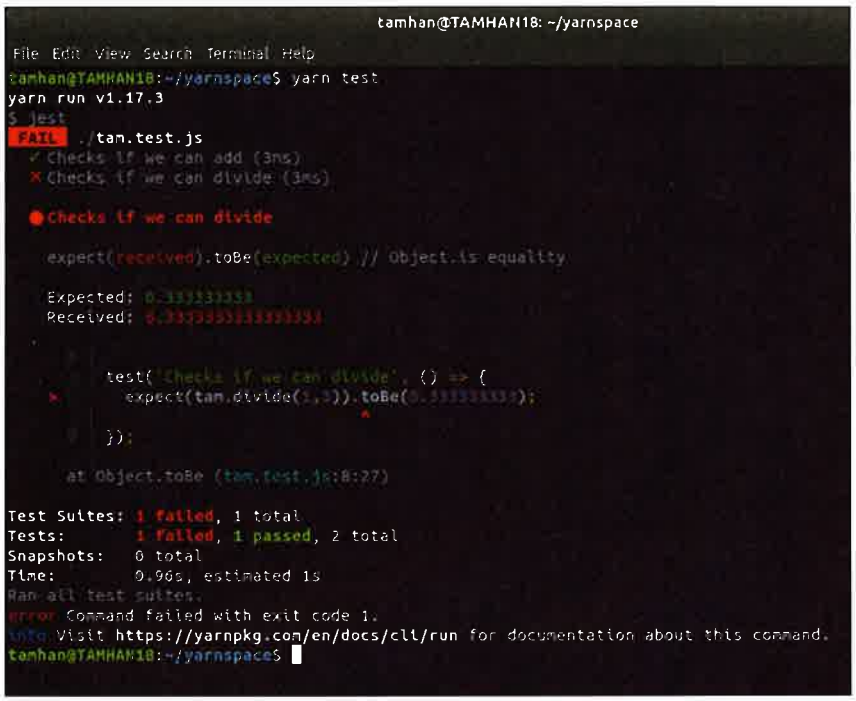


Abb. 2: 0,333333333 ist nicht 0,3333333333333333

*yarn test* einen ersten Probelauf. Die wird mit der in **Abbildung 1** gezeigten Fehlermeldung scheitern.

Wer Jest – wie in der Datei *package.json* – nicht zusätzlich weiter parametriert, aktiviert die im Framework automatisch enthaltene Dateiergreifungsfunktion. Sie nutzt den in **Abbildung 1** in Ocker hervorgehobenen *RegEx*-String, um alle zum Projekt gehörenden Dateien zu durchsuchen. Leider findet unser Programm zum Zeitpunkt unserer Arbeiten keine derartigen Dateien. Der Unterordner, in dem die von Node.js verwendeten Module unterkommen, ist ausgeschlossen.

Über das, was einen guten Demonstrationsfall für ein Unit-Test-Framework ausmacht, lässt sich seit jeher hervorragend streiten. Ich möchte in den folgenden Schritten eine Datei namens *tam.js* anlegen, die im Stammverzeichnis des Jest-Prozesses liegen soll. Öffnen Sie sie in einem JavaScript-Editor Ihrer Wahl und fügen Sie anfangs die drei Methoden – wie in Listing 2 zu sehen – ein.

Neben dem beliebten Zurückgeben eines Strings finden sich hier auch zwei Funktionen, die mathematische Operationen durchführen. Das ist insbesondere in der Welt von JavaScript eine wichtige und undankbare Aufgabe – der Sprachstandard kennt keinen Weg, Fixkommazahlen und natürliche Zahlen voneinander zu unterscheiden. In der Praxis hat man deshalb immer das Problem, wie genau man Zahlen vergleichen möchte. Ob 8,88888 und 8,88889 identisch sind, ist nun mal Stoff für Diskussionen. Schon hier sei übrigens angemerkt, dass der Umfang der numerischen Vergleichsfunktionen ein exzellentes Werkzeug zur Überprüfung der Qualität und einer bequemen Bedienbarkeit eines JavaScript-Test-Frameworks darstellt.

Speichern Sie *tam.js* danach auf Ebene der Datei *package.json*. Leider sind wir damit noch nicht komplett am Ziel angekommen. Für einen erfolgreichen Durchlauf von Yarn *test* benötigen wir nach wie vor eine Testdatei, deren Name nach dem Schema ***/__tests__/**/*.[jt]s?(x),**/?(*.)+(spec|test).[tj]s?(x)* aufgebaut sein muss. In den folgenden Schritten wollen wir deshalb auf die Datei *tam.test.js* setzen, die wir neben *tam.js* platzieren und mit folgendem Inhalt ausstatten:

```
const tam = require('./tam');

test('Checks if Tam is friendly', () => {
  expect(tam()).toBe("Hello");
});
```

Wer in der Vergangenheit mit anderen Test-Frameworks für JavaScript gearbeitet hat, findet an dieser Stelle im Großen und Ganzen Bekanntes. Auch die für alles andere vorgesehenen Testfälle müssen im globalen Namespace der Datei vorliegen; die *test*-Funktion nimmt neben einem beschreibenden String auch einen Funktionszeiger auf, der die eigentlich zu erledigende Aufgabe bearbeitet. In der Testfunktion finden sich dann diverse Matcher, mit denen man die Korrektheit der vom Testcode zurückgelieferten Informationen gegen die vom Entwickler bereitzustellenden Konstanten überprüft.

Interessant ist, dass die Testlogik nun über die *Require*-Funktion in die Datei geladen wird. Das ist eine Anpassung an die sonstige Vorgehensweise innerhalb der Yarn-Dokumentation: Da das Framework ausschließlich für den objektorientierten Einsatz vorgesehen ist, sind so gut wie alle Beispiele auf Basis von *Require* aufgebaut. Es spricht in der Praxis allerdings nichts dagegen, die Logik auf eine andere Art und Weise in den Scope der Testdatei zu bringen. An dieser Stelle können wir abermals *yarn test* eingeben, um einen ersten Testlauf des Codes zu befehlen:

```
tamhan@TAMHAN14:~/yarnspace$ yarn test
yarn run v1.13.0
$ jest
 PASS ./tam.test.js
```

Eine aufmerksame Betrachtung der Kommandozeilenausgabe zeigt uns, dass der Befehl *yarn test* im ersten Schritt den in der Jest-Ausführungsumgebung enthaltenen Runner anwirft. Der ruft danach auf Kommandozeilenebene *jest* auf, um das eigentliche Test-Framework loszulassen. Da die in der Datei *tam.test.js* deklarierte Bedingung von der Funktion erfüllt wird, erfolgt die Ausgabe des Parameters *Pass*. Die Ausgabe *error Command failed with exit code 1.* fehlt ersatzlos, ein eventuell als Aufrufer agierendes Skript würde an dieser Stelle den Rückgabewert *0* erhalten.

Als nächste Aufgabe möchte ich die weiter oben angesprochenen Probleme mit der Fließkommaarithmetik näher betrachten. Da unser Jest-Code auf dem Modulpattern basiert, müssen wir hierzu im ersten Schritt in *tam.js* zurückkehren und die Additions- und Divisionsfunktionen ebenfalls exponieren:

```
module.exports.add = add;
module.exports.divide = divide;
```

Als Nächstes folgen dann zwei Unit-Tests, die die korrekte Division und korrekte Addition überprüfen. Der Autor hat die Vergleichswerte im Interesse der Lustigkeit mit seinem TI-84 ermittelt, dessen Genauigkeit aufgrund eines Kunden ein wenig eingeschränkt ist:

```
const tam = require('./tam');
test('Checks if we can add', () => {
  expect(tam.add(2,2)).toBe(4);
});

test('Checks if we can divide', () => {
  expect(tam.divide(1,3)).toBe(0.333333333);
});
```

Von besonderem Interesse ist, dass die Additionsfunktion mit natürlichen Zahlen arbeitet. Die Divisionsmethode muss stattdessen mit Fließkommazahlen auskommen, was die Genauigkeitsanforderungen erhöht bzw. verkompliziert. Zur Überprüfung der Situation wollen wir an dieser Stelle einen weiteren Kompilations-Run befähigen, der mit dem in **Abbildung 2** gezeigten Fehler scheitert.

Als theoretische Lösung des Problems bietet sich eine Subtraktion des Ziel- vom Sollwert an. Sofern man danach den Absolutbetrag der Differenz ermittelt, lässt sich bis zu einem gewissen Grad abschätzen, ob die Routine korrekt funktioniert. Dieser Prozess artet in der Praxis schnell in Arbeit aus, weshalb das Jest-Entwicklerteam für Vergleiche von Fließkommazahlen einen besonderen Matcher spendiert. Unter seiner Verwendung sieht unser Testfall so aus:

```
test('Checks if we can divide', () => {
  expect(tam.divide(1,3)).toBeCloseTo(0.333333333);
});
```

Wenn wir die Datei *tam.test.js* speichern und die Testfallbatterie abermals zur Ausführung freigeben, stellen wir fest, dass nun alles ohne Probleme funktioniert. Der Manager übernimmt auf Wunsch auch einen weiteren Parameter, über den Sie die beim Vergleich anzuwendende Genauigkeit festlegen können. Wer besonderes Interesse an den dahinterstehenden mathematischen Prozessen hat, findet in der StackOverflow-Diskussion unter [2] Rat und Hilfe.

```
  console.log tam.test.js:20
    beforeAll

  console.log tam.test.js:12
    beforeEach

  console.log tam.test.js:16
    afterEach

  console.log tam.test.js:12
    beforeEach

  console.log tam.test.js:16
    afterEach

  console.log tam.test.js:24
    afterAll

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        1.471s
Ran all test suites.
error Command failed with exit code 1.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this comm
and.
```

Abb. 3: Die Reihenfolge erscheint unter der Ausgabe des Test-Runners

### Eine Frage der Ablaufsteuerung

Unit-Tests sollten – so zumindest die ursprüngliche Definition – mehr oder weniger alleinstehende Testfälle sein. In der Praxis stellte die Entwicklerschaft schnell fest, dass es empfehlenswert ist, die einzelnen Testfälle in eine Art Lebenszyklus einzubinden. Hinter diesem auf den ersten Blick kompliziert klingenden Gedanken steht das Konzept, dass die Testklasse im Rahmen ihrer Initialisierung und/oder Abtragung beispielsweise Ressourcen anlegen oder Speicher allozieren kann. Sinn dieser Vorgehensweise ist neben einer wesentlichen Beschleunigung der Testausführung durch Reduktion der elektronischen Last auch, den in den einzelnen Testfällen befindlichen Code zu verkürzen und so übersichtlicher zu gestalten.

Wenn Ihre Testfälle dabei eine enge Beziehung zu den zugrunde liegenden Ressourcen eingehen, ist es empfehlenswert, die Abtragung und Errichtung für jede einzelne Testbedingung separat durchzuführen. In diesem Fall greifen Sie auf das Methodenpaar *beforeEach* und *afterEach* zurück:

```
beforeEach(() => {
  initializeCityDatabase();
});

afterEach(() => {
  clearCityDatabase();
});
```

In der Praxis gibt es auch Situationen, in denen ein breiterer Scope der Ressourcen völlig ausreicht. In solchen Fällen können wir stattdessen die Methoden *beforeAll* und *afterAll* verwenden. Initialisierung und Abtragung erfolgen dann weniger häufig, was den Ressourcenverbrauch reduziert:

```
beforeAll(() => {
  return initializeCityDatabase();
```

```
tamhan@TAMHAN18:~/yarnspace$ yarn test
yarn run v1.17.3
$ jest
 FAIL  ./tam.test.js
  ✕ this will be the only test that runs (5ms)
  ○ skipped Checks if we can add
  ○ skipped Checks if we can divide

  ● this will be the only test that runs

    expect(received).toBe(expected) // Object.is equality

    Expected: false
    Received: true

      12 |   test.only('this will be the only test that runs',
      13 |   () => {
    > 14 |     expect(true).toBe(false);
         |                  ^
      15 |   });

      at Object.toBe (tam.test.js:14:18)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 2 skipped, 3 total
Snapshots:   0 total
Time:        1.073s
Ran all test suites.
error Command failed with exit code 1.
```

Abb. 4: „test.only" deaktiviert alle anderen Testfälle

```
});

afterAll(() => {
  return clearCityDatabase();
});
```

Im Interesse der besseren Visualisierung des Lebenslaufs wollen wir die vier Methoden im nächsten Schritt in unsere Testdatei einfügen. Da wir unser Programm sowieso im Konsolenfenster ausführen, können wir nach dem folgenden Schema die Ausgabe von Statusinformationen befehligen. Wir drucken hier aus Platzgründen nur den Korpus der Methode *afterAll* ab – die anderen drei Methoden sollten Sie (bis auf die Verwendung von anderen Strings) analog parametrieren:

```
afterAll(() => {
  console.log("afterAll");
});
```

Lohn der Ausführung ist die in **Abbildung 3** gezeigte Liste von Ausgaben. Der Jest Runner sammelt die Konsolenausgabe während der Programmausführung ein, um sie danach en bloc unter den Statusinformationen auszugeben. Beachten Sie zudem, dass das Entwicklerteam das Verhalten von *console.log* immer wieder anpasst. Die unter [3] zu findende Diskussion sollte in Ihrem Browser als Favorit unterkommen, wenn Ihre Jest-Testfälle häufig Daten in die Kommandozeile schreiben.

Ein weiterer, in diesem Zusammenhang sehr interessanter Aspekt ist die Möglichkeit, durch Aufruf von *test.only* einen Test als „alleinstehend" zu markieren. Findet das Framework eine entsprechende Methode, ignoriert es alle anderen Testfälle. Als Beispiel dafür wollen wir die folgende Struktur in die Datei *tam.test.js* einbinden:

```
test.only('this will be the only test that runs',
() => {
  expect(true).toBe(false);
});
```

Wenn Sie die Ausführung dieser Datei befehligen, sehen Sie das in **Abbildung 4** gezeigte Resultat.

Jest parst alle Dateien, bevor die Ausführung der Testbatterie beginnt. Ein mit *only* markierter Test deaktiviert alle gewöhnlichen Kollegen, und führt am Bildschirm zur Ausgabe von Skipped-Statusmeldungen. Die kurzfristige Vereinzelung von Testfällen lohnt sich insbesondere dann, wenn Ihr Projekt eine vergleichsweise umfangreiche Testsuite mitbringt. Auf diese Art und Weise lässt sich ein neu hinzugefügtes Feature fokussiert überprüfen, ohne viel Zeit mit der Ausführung der Testbatterie zu verlieren. Dass diese Vorgehensweise naturgemäß die Wahrscheinlichkeit von Regressionen und sonstigen versehentlich verursachten Fehlern erhöht, sei im Interesse der Vollständigkeit allerdings angemerkt.

## Objektorientierte Analyse

Objektorientierte Programmierung und JavaScript galten lange Zeit als eine sehr ungünstige Kombination. Das lag unter anderem daran, dass die Sprachspezifikation OOP nur sehr leidlich vorsah und durch diverse Designpattern zur OOP-Emulation gezwungen werden musste. Das von Addy Osmani veröffentlichte Buch stach an dieser Stelle insofern aus der Masse heraus, als es die Popularität derartiger Emulationsansätze wesentlich erhöhte. In der heutigen Zeit gibt es kaum einen Entwickler, der diese Designpatterns nicht im Hinterkopf hat und bei der täglichen Arbeit automatisch anwendet. Zudem brachten neue Versionen von ECMAScript Convenience-Methoden und Syntactic Sugar mit, die Entwicklern die Implementierung von Klassen erleichtern. Daraus ergibt sich allerdings auch eine neue Situation für Test-Frameworks. Klassische Vergleiche der Objektinhalte sind insofern kompliziert, als die Nutzung des ==-Operators fehlschlägt. Der liefert ja nur dann *True* zurück, wenn beide Variablen auf dieselbe Objektinstanz (also nicht nur auf ein identisches Objekt) verweisen. Möchte man Objekte auf Member-Ebene vergleichen, so blieb bisher nur der Umweg über manuelle Vergleichslogik.

Das wichtigste Feature von Jest sind die in der Einleitung vorgestellten Snapshots. Ein Snapshot ist – grob gesagt – eine Beschreibung eines Objekts, die die für einen Vergleich relevanten Attribute benennt und auch die Sollwerte mitliefert. Die in Jest implementierte Vergleichs-Engine kann diese auf eine vom Code angelieferte Objektinstanz anwenden, um das Vorhandensein bzw. die Korrektheit der im Objekt enthaltenen Attribute sicherzustellen. Was auf den ersten Blick immens

kompliziert klingt, erweist sich in der Praxis als immens wertvoll. Die Snapshots haben die Eigenschaft, wesentlich kürzer zu sein als eine von Hand auf Basis von *If, Else* und Co. aufgebaute Selektion.

Ein besonders netter Aspekt der Implementierung ist, dass die Erzeugung der Strukturen bis zu einem gewissen Grad vom Framework erledigt wird. Im ersten Schritt beginnen wir damit, den folgenden Testfall in einer neuen Testdatei anzulegen. Im Interesse der Bequemlichkeit empfiehlt es sich an dieser Stelle, die bisher verwendete Datei in Sicherheit zu bringen. Ihre Ausführung würde nur für zusätzlichen Clutter am Bildschirm sorgen:

```
it('is a small test', () => {
  const user = {
    createdAt: new Date(),
    id: Math.floor(Math.random() * 20),
    name: 'Tam HANNA',
  };

  expect(user).toMatchSnapshot();
});
```

Technisch gesehen ist das keine Hexerei. Wir erzeugen einen aus drei Werten bestehendes Objekt. Neben einem Datum weisen wir auch einen fixen Namen und eine zumindest pseudozufällige Zahl zu. Danach rufen wir die Funktion *toMatchSnapshot* auf, einen Matcher, der die Objektinstanz mit einem Snapshot vergleichen soll. Mitdenkende Leser fragen sich bei der Ausführung dieses Testfalls, wo das Test-Framework die zum Vergleich erforderlichen Daten hernehmen soll. Die Antwort findet sich in der Kommandozeile, wo wir bei der erstmaligen Abarbeitung des Befehls das in der **Abbildung 5** gezeigte Ergebnis bekommen.

Im nächsten Schritt suchen wir nach dem Ordner *__snapshots__*, in dem wir eine Datei namens *tam.test.js.snap* finden. Ihr Inhalt sieht anfangs wie folgt aus. Wundern Sie sich nicht, wenn die auf Ihrer Workstation aufscheinenden Ergebnisse ob der Zufälligkeit etwas anders aussehen:

```
exports[`is a small test 1`] = `
Object {
  "createdAt": 2019-08-12T00:40:13.198Z,
  "id": 10,
  "name": "Tam HANNA",
}
`;
```

Spaßeshalber bietet es sich an, an dieser Stelle noch einen oder zwei Durchläufe der Testsuite anzuweisen. Jest findet ja nun eine Snapshot-Datei, und vergleicht diese mit den in der neu generierten Instanz befindlichen Werten. Wegen der Zufälligkeit ist es sehr wahrscheinlich, dass dieser Vergleich fehlschlägt – von Haus aus werden nämlich alle Attribute eins zu eins miteinander verglichen.

```
tamhan@TAMHAN18:~/yarnspace$ yarn test
yarn run v1.17.3
$ jest
 PASS  ./tam.test.js
  ✓ is a small test (5ms)

 › 1 snapshot written.
Snapshot Summary
 › 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 written, 1 total
Time:        1.09s
Ran all test suites.
Done in 1.74s.
tamhan@TAMHAN18:~/yarnspace$
```

Abb. 5: Jest kündigt das Auffinden neuer Objekte und die Erzeugung neuer Snapshots an

So bequem die automatische Erzeugung von Snapshot-Dateien und ihre manuelle Bearbeitung auch wäre, so unproduktiv ist das in der Praxis. Stellen Sie sich vor, dass Entwickler A umfangreiche Änderungen in der Datei vornimmt und Entwickler B einige Zeit später zusätzliche Attribute aufbietet, was über kurz oder lang zu Inkonsistenzen in der vorliegenden Codebasis führt. Jest umgeht dieses Problem insofern, als die in der Testfalldefinition vorliegenden Informationen immer die Primatrolle übernehmen.

Die zuvor zum Vergleichen von gewöhnlichen Werten verwendeten Matcher lassen sich auch gegen Objektattribute anwenden. Öffnen Sie die Testdatei abermals, und ersetzen Sie den bisher dort befindlichen Code durch den aus Listing 3.

Die beiden Attribute *createdAt* und *id* werden nun durch den Matcher *expect.any* überprüft. Er nimmt statt eines spezifischen Werts einen Typparameter entgegen und akzeptiert zur Laufzeit alle Werte, die vom Datentyp her mit dem angelieferten Parameter übereinstimmen.

Natürlich dürfen Sie auch andere Matcher verwenden, um spezifische Wertevergleiche durchzuführen. Nicht mit einem spezifischen Matcher ausgestattete Parameter scheinen nach wie vor eine 1:1-Überprüfung vorzunehmen.

Wer den Code an dieser Stelle ausführt, bekommt abermals einen Fehler vorgesetzt. Die Ursache dafür ist, dass Jest aus Performancegründen bei schon vorhandenen Objekten keinerlei Überprüfungen der Aktualität des Snapshots vornimmt. Wir müssen deshalb im ersten Schritt von Hand eine Aktualisierung der Snapshot-Dateien bewilligen:

```
tamhan@TAMHAN18:~/yarnspace$ yarn test -u
yarn run v1.17.3
$ jest -u
...
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 updated, 1 total
```

**Listing 3**

```
it('is a small test', () => {
  const user = {
    createdAt: new Date(),
    id: Math.floor(Math.random() * 20),
    name: 'Tam HANNA',
  };

  expect(user).toMatchSnapshot({
    createdAt: expect.any(Date),
    id: expect.any(Number),
  });
});
```

**Wer asynchronen Code verwendet oder mit Objekten zu tun hat, kann mit Jest viel Zeit sparen.**

Die Auswirkungen davon lassen sich analysieren, wenn Sie die Datei *tam.test.js.snap* abermals in einem Texteditor Ihrer Wahl öffnen. Ihr Korpus präsentiert sich nun folgendermaßen:

```
exports[`is a small test 1`] = `
Object {
  "createdAt": Any<Date>,
  "id": Any<Number>,
  "name": "Tam HANNA",
}
`;
```

Nach der erfolgreichen Aktualisierung der Snapshot-Datei können Sie mehrere Testläufe durchführen. Da sowohl die Zahl als auch das Datum immer den gleichen Datentyp aufweisen, sollte es fortan zu keinen Fehlern bei der Ausführung des Testcodes mehr kommen.

### Und jetzt asynchron

Während man die bisher realisierten Funktionen – mit Zusatzaufwand – auch von Hand realisieren könnte, zeigen klassische JavaScript-Test-Frameworks spätestens beim Durchführen von Testoperationen gegen asynchronen Code massive Probleme. Die Ursache dafür ist, dass im Test-Runner normalerweise keine Methode vorgesehen ist, um die Ausführung auf das Ergebnis der asynchronen Aufgabe warten zu lassen.

Wegen seiner späten Geburt ist Jest auch an dieser Stelle in der Lage, für Entwickler Abhilfe zu schaffen. Als erstes kleines Beispiel wollen wir uns den folgenden Testfall ansehen, der eine primitive asynchrone Operation realisiert:

```
test('the data is peanut butter', done => {
  function callback(data) {
    expect(data).toBe('peanut butter');
    done();
  }

  fetchData(callback);
});
```

Die Verwendung des Arguments *done* an Stelle des normalen Funktions-Pointers informiert die Runtime darüber, dass der im Rahmen dieses Testfalls ablaufende Code asynchron ist. Daraus folgt, dass Jest während der Testausführung einige Zeit wartet, ob der *done* Callback aufgerufen wird. Passiert das, gilt der Test als abgeschlossen, bleibt die Meldung während der (in der Dokumentation nicht genannten) Wartezeit aus, gilt der Test als gescheitert.

Wer mit Promises arbeitet, findet einen bequemeren Weg in der direkten Verarbeitung des Rückgabewerts. Als Beispiel hierfür wollen wir abermals von der Funktion *fetchData()* ausgehen, die nun aber ein Promise retourniert:

```
test('the data is peanut butter', () => {
  return fetchData().then(data => {
    expect(data).toBe('peanut butter');
  });
});
```

Interessant ist, dass Testfälle auch auf scheiternde Promises parametriert werden können. Der hierzu notwendige Code sieht folgendermaßen aus:

```
test('the fetch fails with an error', () => {
  expect.assertions(1);
  return fetchData().catch(e => expect(e).toMatch('error'));
});
```

An dieser Stelle müssen wir unsere Überlegungen zu Jest – schon aus Platzgründen – beenden. In der bereitstehenden Dokumentation [4] findet sich eine detaillierte Besprechung der asynchronen Testprozesse – es gibt kaum einen Testfall, den Jest nicht abdeckt.

### Lohnt es sich?

Außer Frage steht, dass die in Jest enthaltenen Unterstützungsfunktionen für ein sehr komfortables Arbeitserlebnis sorgen. Wer asynchronen Code verwendet oder mit Objekten zu tun hat, spart bei intelligenter Nutzung des Frameworks viel Zeit.

Ob man in der Praxis ein schon vorhandenes Projekt umstellen sollte, ist allerdings eine wesentlich schwierigere Frage. Erstens ist das Umschreiben von Unit-Tests eine sehr undankbare Aufgabe, zweitens müssen Sie immer überprüfen, ob das Entwicklerteam gewillt ist, die Änderungen am Testfall-API auf sich zu nehmen ...

**Tam Hanna** befasst sich seit der Zeit des Palm IIIc mit der Programmierung und Anwendung von Handcomputern. Er entwickelt Programme für diverse Plattformen, betreibt Onlinenewsdienste zum Thema und steht unter tamhan@tamoggemon.com für Fragen, Trainings und Vorträge gern zur Verfügung.

### Links & Literatur

[1] https://yarnpkg.com/lang/en/docs/install/
[2] https://stackoverflow.com/questions/50896753/jest-tobeclosetos-precision-not-working-as-expected
[3] https://github.com/facebook/jest/issues/2441
[4] https://jestjs.io/docs/en/asynchronous.html