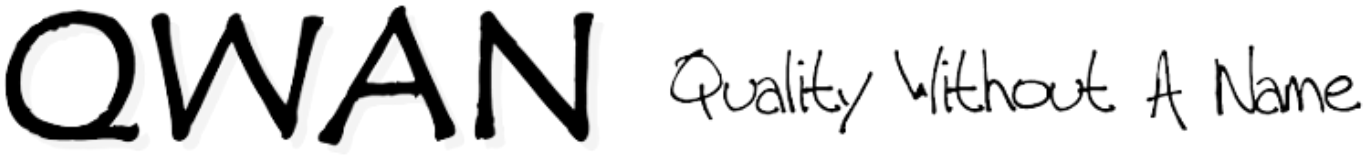
Property Based Testing Hands On - Javascript introduction



April 2016

- Rob Westgeest - rob@qwan.eu
- Marc Evers - marc@qwan.eu
- Willem van den Ende - willem@qwan.eu

Prerequisites

You need an environment with Javascript and Mocha/Chai (Node) to do the Javascript exercise. We use the **jsverify** library in this tutorial. There are many libraries available in your favorite languages, but the basic principles are the same.

```
https://github.com/jsverify/jsverify
```

You need the following node modules for these exercises (which have already been installed if you are using the provided cyber-dojo enviroment):

```
npm install jsverify
npm install lodash
```

Property based testing concepts

We will introduce the following concepts - **properties** - functions that capture invariants of the system under tets - **arbitraries**, **generators** and **shrinking** - functions that produce arbitrary input data and minimal counter examples - restricting input values

First steps at properties and property based testing

Let's start with a simple example using some math functions, to introduce the basic concepts. An invariant of the square root and square functions is:

- a number's square is equal to that number: sqrt(x * x) = x

In an example based test approach, we would write a small number examples showing that this holds. We now want to 'proof' that the mentioned invariant holds for all numbers.

Create a new file firstTest.js with:

```
"use strict";

// Import the JSVerify library:
var j = require("jsverify");

// Define the 'system under test':

function squared(x) {
  return x * x;
}
```

We want to show that for all numbers, the following holds: Math.sqrt(squared(n)) === n, so we define a *property* by adding to the file:

```
var squareRootOfNSquaredEqualsN = j.forall(j.integer, function (n) {
  return Math.sqrt(squared(n)) === n;
});
```

We will use Mocha & Chai to embed the property based tests into a testing framework:

```
var options = { tests: 100 };

describe('Square root', function () {
  it('square root of n squared always equals n', function () {
    j.assert(squareRootOfNSquaredEqualsN, options);
  });
});
```

You can use the options to vary the number of samples generated. If you leave it out, the default is 100.

Run the tests.

Alternatively, you can run the property based tests without a testing framework (run it with node firststeps.js):

```
var options = { tests: 100, quiet: false };

j.check(squareRootOfNSquaredEqualsN, options);
```

Run the file with: node firstTest.js

What happens?

JSVerify generates input test data and checks if the property holds (returns true) for all the generated input values. If it finds a case for which the property does not hold, it fails and returns the counter example.

Currently it is configured to generate 100 test cases. You can specify a different number using the tests property of the options.

How does it know to generate test data? j.integer() does the job here. It is an *arbitrary* that can generate random integers. More about generators later on.

The test fails because the invariant does not hold for negative numbers. We should restrict the generated input to natural numbers only (>=0). Replace j.integer() by j.nat() and run it again.

JSVerify provides arbitraries for all the basic types:

- j.integer - integers
- j.nat - natural numbers
- j.number - doubles
- j.string - strings (can be empty, can contain non-ascii characters)
- j.asciistring - strings with ascii character

Another property

Now, define another simple property, for getting the feel of it. Define as an invariant that the square of a number is always greater than (or equal to?) that number.

You can add an extra property definition to the same file as well as an extra it/assert check-call to the describe block.

```
var squareNIsGreaterThanN = ...

describe( ...
...
  it('...', function () {
    j.assert(squareNIsGreaterThanN, options);
  });
```

Try it out. Play with the conditions to make it fail on purpose and look at the counter examples that are reported.

What we have learned

We have learned about *properties* which are invariants of the code under test and about *arbitraries* that generate random, valid inputs for the code under test.

Arbitraries vs generators: in texts about property based testing, you will encounter both 'arbitraries' and 'generators'. A generator is a function that can generate arbitrary data of a specific type. An arbitrary combines a generator with a shrinking function. Shrinking is used when a counter example is found, to reduce the counter example to the smallest one that still fails the property.

A more complicated exercise

Let's put our teeth in a slightly more interesting example, to learn more about property based testing.

We want to create a smart sorting/grouping feature for restaurant menus. Our input is a randomly ordered list of dishes, each having a course. for example:

```
[
  {course: "starters", dish: "pomodoro soup"},
  {course: "main dish", dish: "salmon"},
  {course: "starters", dish: "veggy soup"},
  {dish: "nuts"},
  {course: "main dish", dish: "cannelloni"}
]
```

We want to sort and order this mess so that: - dishes are grouped by course - dishes without course are put at the end and get the course "other" - groups of dishes should be put in this specific order: "starters", "main dish", "side dish", "other" - dishes are sorted alphabetically within the groups

First property: a sorted result

Let's build our sorting & grouping function in small steps, property by property. Start with a stubbed function (in a new file menuSortTest.js):

```
"use strict";

var j = require("jsverify");
var options = { tests: 100 };

function menuSort(items) {
  return [];
}

// start with the sorting of items, first just a list of strings:

var dishesAreSorted = j.forall(j.nearray(j.asciistring),
  function (menuItems) {
    var sorted = menuSort(menuItems);
```

```
  // use the reduce function on array to compare consequtive pairs
    return sorted.reduce(function (isSortedUntilHere, current, index, array) {
      return isSortedUntilHere &&
        (index === 0 || array[index].localeCompare(array[index-1]) > 0);
    }, true);
  });
```

Add a check call to verify this property. What happens? Implement the sorting function:

```
function menuSort(items) {
  return items.sort(function (a,b) {
    return a.localeCompare(b);
  });
}
```

Run the test again. What happens now? Make it work!

We are using the j.nearray combinator for arbitraries. Based on a provided arbitrary (asciistring for ASCII strings in this case), it returns a new arbitrary that generates arrays of ASCII strings.

More interesting arbitraries

We don't want plain strings as inputs, we want to work with more structured objects instead. How do we generate those?

We already saw how to compose an arbitrary from an array arbitrary and a string arbitrary. There are more combinators available.

```
var generateMenuItem = j.record({course: j.asciistring, dish: j.asciistring})
```

It is a good idea to extract the course and dish generators as well:

```
var generateCourse = j.asciistring;
var generateDish = j.asciistring;
var generateMenuItem = j.record({course: generateCourse, dish: generateDish});

var generateMenu = j.nearray(generateMenuItem);
```

The record combinator generator generates JSON objects according to the 'specs'. In this case, we generate objects with a course property of type string and a dish property of type string.

Don't forget to update the sorting function: a.dish.localeCompare(b.dish)

and the property checks: array[index].dish

Refining the property and the arbitrary

The current definition of the property checks if all menu items are sorted, but we want to have them sorted per course. Let's group the sorted menu items by course and check per group of menu items whether they're sorted.

```
// 1. We're going to use some useful stuff from the Lodash library,
// add it at the start of the file:
var _ = require("lodash");

// 2. Extract an 'isSorted' function that checks if an array of menuItems is sorted:
function isSorted(menuItems) {
  return menuItems.reduce(function(isSortedUntilHere, current, index, array) {
          return isSortedUntilHere && (index === 0 ||
array[index].dish.localeCompare(array[index-1].dish) >= 0);
        }, true);
}

var valuesAreSortedPercourse = j.forall(j.nearray(menuitems), function(values) {
      var sorted = menuSort(values);

// 3. Use 'groupBy' from Lodash to get a json object with an array per value of
property "course":
      var sortedPerCourse = _.groupBy(sorted, "course");

// 4. Use the  'values' function to get an array of all the
// arrays, and for each one we can check it is sorted:
      return _.values(sortedPerCourse).every(isSorted);
});
```

Run the tests; does it work?

Do you think the sortedPerCourse should be part of the domain code (menuSort)?. And if so, do you think the name 'menuSort' is still appropriate?

Conditional properties

Usually, not all inputs are sensible or valid. We want to restrict checking properties to valid input.

We only want to use non-empty strings for menu items. We could solve this by adding an extra filtering condition to our property that short-circuits the property to true for invalid input values:

```
// Let's keep count of the number of inputs we're dropping, so add this
line somewhere at the start of the file:
var droppedInputs = 0;

// ...
// Add a validity check to the property:
  if (menuItems.some(function (menuItem) { return menuItem.dish === ""; })) {
    droppedInputs = droppedInputs + 1;
    return true;
```

```
  }

// ...
// Log the number of dropped inputs to the test, after the assert on the property:
  console.log('Dropped inputs: ' + droppedInputs);
```

Run the tests. Observe what happens.

The disadvantage of filtering input data is that you drop quite a lot of data and your test becomes less meaningful because of the low number of actual samples tested.

An alternative approach is to put restriction on your arbitraries. JSVerify offers the 'suchthat' function to restrict generated values:

```
var generateDish = j.suchthat(j.asciistring, function (s) { return s.length > 0;
});
var generateCourse = j.elements(["main", "drinks", ""]);
var generateMenuitem = j.record({course: generateCourse, dish: generateDish});
```

Run the tests and observe what happens. How many inputs are dropped now? Remove the input check from the property.

The 'elements' arbitrary generates values from a given array.

Defining more properties

Property based testing becomes more fun when we actually get to define and check more properties of the system under test. Let's define another one for our sorter/grouper.

Add a property that checks that for every menu item in the sorted result, the course is not empty. Write the property. Make it work by refining the sorting function.

Another property is whether the menu items are grouped by their courses. How can you check this? Write the property.

Modelling Money

In this exercise, we are going to build a class representing Money, step by step. Representing money in software might sound easy, but it is actually non trivial and is often done wrong. Doing it wrong can result in small or big rounding errors or just losing a few millions on a bad day...

Getting started

Create a new file (moneyTest.js), with the jsverify boilerplate:

```
"use strict";
```

```
var j = require("jsverify");
var _ = require("lodash");

var options = { tests: 500 };
```

Create a class to represent money. It is constructed with an amount and a currency.

```
function Money(amount, currency) {
  this.amount = amount;
  this.currency = currency;
}

Money.prototype.add = function (money) {
  // ...
};
```

Start with something simple: when we add two amounts of the same currency, we get a new amount. Our property needs two money objects, so we pass two arbitraries to forall and define a property function with two arguments:

```
var moneyOfSameCurrencyAddsUp = j.forall(generateAmount, generateAmount,
function(a1, a2) { ...
});
```

Create the *generateAmount* arbitrary. We can generating the currency e.g. using:

var generateCurrency = j.elements(["EUR", "USD", "GBP"]);

How do we generate Money instances? We use the *pair* arbitrary combinator to generate amount-currency pairs and use jsverify's *smap* function to transform the pair into a Money object (and back again). 'smap' stands for symmetric map, and requires two functions, to do a mapping and to perform the inverse mapping:

```
var generateAmount = j.pair(j.nat(), generateCurrency).smap(
  function (x) { return  new Money(x[0], x[1]); },
  function (money) { return [money.amount, money.currency]; } );
```

Now we can generate Money objects! For now we use natural number for the amount.

Why do we need to provide a function to convert a money object back to its components? JSVerify needs the inverse for *shrinking* counter examples: if a counter example is found, jsverify will try to shrink the counter example to the smallest example that recreates the failure. This will help in better understanding the counter example.

Now add the behaviour that Money can only be added to money of the same currency. Throw an exception when the currencies are different. Make sure your property handles this correctly.

```
var incompatibleCurrencies = {
  name: 'cannot add money from different currencies',
  toString: function () { return this.name; }
}

Money.prototype.add = function (money) {
  if (this.currency !== money.currency) {
    throw incompatibleCurrencies;
  }
  // ...
};
```

Add a new property that checks that adding is not possible whenever the currencies are different.

Money allocation

We want to be able to allocate money: divide it in equal parts, without money getting lost in rounding. An example: dividing €100 in 3 should yield [€33, €33, €34] (assuming whole numbers for now).

Define a property that states that no money gets lost when dividing an amount.

Note that we should make the integer assumption of the amounts more explicit, otherwise Javascript will just use doubles.

```
function Money(amount, currency) {
  this.amount = Math.floor(amount);
  ...
```

Converting from string

We want to be able to create money objects from strings; we support EUR and USD. Some legal values: - 100 (default is EUR) - EUR 10 - USD 300 - USD 40- (negative value) - EUR 10.000 (. as thousand separator for euros)

And some illegal values:

- USD 4.999 (USD has , as thousand separator and . as cents separator)
- 1,000 (illegal because EUR is default and EUR uses . as thousand separator)
- HFL 50 (illegal currency)
- EUR 10bla

In our domain model, we work with valid objects. This part focuses on processing and validation outside, 'untrusted' data that is transformed into domain objects (e.g. data coming in from a web form). We want this transformation to be robust, so that it can handle any data and only results in valid domain objects when it is able to (and we don't need to do defensive programming in our domain).

There are actually two concerns we need to address:

- Given the input string is valid, will our code create a valid Money object with the correct contents? (correctness)
- Is the conversion code robust for every possible input string? The code should always either return a valid Money object or a rejection of the input. It should not throw unexpected exceptions or create invalid Money objects (robustnesss)

Let's capture these in properties.

Define conversion function that returns a JSON object that is either { money: } or { error: 'some error message' }

Start e.g. with something like:

```
function moneyFromString(input) {
  var theMoney = new Money(...);
  return { money: theMoney };
}
```

We can start with either correctness or robustness first. It would be interesting to try both approaches and see whether and how it drives you to different design decisions.

Correctness

Given correct input, the conversion should produce valid Money objects. How would you define this as property?

Write an arbitrary that creates valid input strings. You can again use smap to create valid strings from primitive values.

Hints and tips: - If you let your arbitrary generate a tuple (array) of an input string together with the corresponding amount and currency values, it will be easier. for your property function to check correctness: **['EUR 100', 'EUR', 100]** - Take baby steps; start e.g. with regular input with a currency; add support for the default currency; add support for thousand separators. - Make sure your test fails for the right reason at every step.

Robustness

Given any input, the conversion should always produce either a valid Money object or a validation error: { money: ... } or { error: 'some message' }

Define a property that captures this.

Write an arbitrary that creates all kinds of input strings. Is the 'asciistring' arbitrary suited for this? Why (not)?

Corner cases that almost look like valid input are particularly interesting, like "EUR 12jsde", "HFL 30", and "300 EUR". Adapt your arbitrary so that it will generate cases like these.

Refine your conversion function and make sure the correctness property is also satisfied.

Hints and tips: - jsverify provides the 'oneof' function that combines two arbitraries into one that generates values taking values from both arbitraries.

Reflection

How did defining the properties and the arbitraries influence you in thinking about the problem?

Where would you like to put the property logic? To what extent is it test code, to what extent should/could it be part of production code?

What happens if you would start with robustness first and then correctness? Do the exercise again, but start with robustness and see if this leads you to different design decisions.

A vending machine with a coin box

In this exercise, we are focusing on a (drinks) vending machine, and specifically the cash register part of such a machine.

Start with modelling a coin box, in which customers can insert different coins and after making a selection, the appropriate amount is checkout out. In the end, the coin box should also correcly handle change: any money left over is returned to the customer, using appropriate coins.

Getting started

Let's start simple. We want to create a Coinbox object with 'insert' and 'checkout' functions. We can model the coin box as having two collections of coins: the 'inbox' (coins inserted but not yet processed) and the 'vault' (coins collected by the coin box after doing checkouts. The coin box keeps the coins until it is emptied by the owner. The coins from the vault can also be used to return the correct change, depending on what types of coins are needed.

We can create representations of all kinds of different coins, but it is sufficient for now to just model two or three (like 10 cents and 50 cents).

Start with a few simple invariants, like: - the total value of the coins in the inbox should always be >= 0 - the total value of the coins in the vault should always be >= 0

Create a new .js file and add the jsverify boilerplate. Create the properties one by one, first letting them fail. Write the simplest version of the Coinbox that satisfies a property.

Mutating state: doing checkouts

Now proceed with implementing checkouts. For now, assume that you do a checkout of all the coins present in the inbox.

First, a property concerning checkouts: first, a checkout empties the inbox. Write a property for this, see it fail for the right reason.

Second, a checkout should not modify the total value of the coins in the inbox and vault.

Write the property, see it fail for the right reason. Implement the checkout function for Coinbox.

Observe closely what happens. Do you get failures? Are they what you expect? If not, what happens?

Hints and tips - It might be useful to define toString functions for your Coinbox and Coins. - If you implement the coin box having mutable state, you might run into confusing feedback from jsverify with counter examples you might not understand. If a test fails, jsverify will use the object to shrink and produce a minimal counter example; this does not work correctly if you modify the generated object. How can you fix this?

Inserting coins

Let's add an insert function for the customer to insert one coin.

What property/properties can you define regarding inserting coins?

Code these properties and implement the insert. Make sure all other properties keep on being satisfied!

Possible next steps

If you want to continue with this exercise, your next steps could be:

- Conditional checkout: pass a price to the checkout function and only check out if the inbox contains a sufficient amount of money.
- Refine the checkout so that a specific amount is checked out and the remaining amount is returned as change. How will you make the coin box return the right coins?