Jasmine による Buddy アプリのテスト

2020 年 1 月 8 日

1 はじめに

この文書ではテストフレームワーク Jasmine を Buddy アプリの中で用いてテストを実施する方法について述べる。

1.1 JASMINE と BUDDY アプリ

Jasmine は JavaScript プログラムのテストを記述して実行するためのフレームワークである。
Jasmine の標準的な利用方法を以下に示す。

1. テスト対象となる JavaScript プログラムを用意する。
2. Jasmine の構文を使ってテスト内容を記述した JavaScript プログラム（これをスペックと呼ぶ）を作成する。
3. Jasmine、テスト対象、スペックを<script>タグで読み込んだ HTML ファイル（これをテスト実行環境と呼ぶ）を作成する
4. テスト実行環境の HTML ファイルをウェブブラウザで開いてテストを実行する。

Buddy は React ベースのシングルページアプリを開発するシステムであり、構築された Buddy アプリは固有の HTML ファイルと JavaScript ファイルから構成されている。そのため、Jasmine の標準的な HTML ファイル形式のテスト実行環境は Buddy アプリを対象としたテストを記述して実行する目的には適さない。

Jasmine ではテストの実行方法やテスト結果の表示方法をカスタマイズするための API が提供されている。本文書では、この API に基づいて開発された Buddy 独自のテスト実行環境 BuddyTestRunner の利用方法について述べる。BuddyTestRunner を用いることで、Buddy アプリのテストを Jasmine のテスト構文により記述してアプリ内で実行することができる。

2 準備

Buddy 用テスト実行環境 BuddyTestRunner を利用するための準備手順を以下に示す。

1. Buddy の開発画面でテスト対象のアプリを開いて「ファイル管理」画面に移動する。
2. ファイル管理画面の操作対象が「設計情報」になっていることを確認して、files ディレクトリ配下の javascripts ディレクトリを開く。
3. 次の 2 つのファイルを javascripts ディレクトリにアップロードして配置する。
   - jasmine.js
   - jasmine_buddy.js

   ここで、jasmine.js は Jasmine 本体を構成するファイルであり Jasmine の配布物に含まれるものと同一である。jasmine_buddy.js は BuddyTestRunner を構成するファイルであり本文書と共に配布されている。なお、Internet Explorer 11 を利用する場合は ES6 Promise のサポートを追加するため以下の URL から es6-promise.auto.js をダウンロードして javascripts ディレクトリに配置する。
   - https://cdn.jsdelivr.net/npm/es6-promise/dist/es6-promise.auto.js

3 JASMINE によるテストの記述と実行

この節では Jasmine を用いた Buddy アプリのテストの記述方法および実行方法について述べる。

3.1 テストの記述

Jasmine 構文に基づく Buddy アプリのテストは、スクリーン設計またはサーバ機能設計において作成する JavaScript プログラムの中に記述する。テストの記述方法はスクリーン設計とサーバ機能設計で共通である。なお、BuddyTestRunner はテストを非同期実行するので、サーバ機能のテストを作成する場合はサーバ機能の実行タイプを「非同期実行」に設定する。

例として、スクリーン設計における BuddyTestRunner の利用方法を以下に示す。この例ではテスト全体を myTest という関数の中に記述している。myTest 関数の api 引数と scope 引数にはテスト対象

となるスクリーンプログラムの api オブジェクトおよび名前空間（this）がそれぞれ渡されてくることを想定している（後述）。

```
function myTest(api, scope) {
	var options = {
		timeout: 60000, // 1 分
	};
	return BuddyTestRunner(this, options, function() {
		// ここに Jasmine の構文によるテストを記述する
		describe("some app functionality", function () {
			it("should perform some task", function (done) {
				// …
				done();
			});
		});
	});
}
```

BuddyTestRunner は 3 つの引数を取る関数である。第 1 引数には Jasmine のテスト構文（describe や it など）が配置される名前空間を与える。第 2 引数には以下のプロパティから成るオプションオブジェクトを与える。

- timeout: 非同期実行のテストがタイムアウトするまでの時間をミリ秒単位で指定する。デフォルト値は 5000 ミリ秒（5 秒）。

第 3 引数には関数オブジェクトを与える。この関数の中に Jasmine のテスト構文を用いたテストを記述する。BuddyTestRunner の第 1 引数に this を与えると、第 3 引数の関数の中では Jasmine のテスト構文があらかじめ用意された状態に設定される。

3.2 テストの実行

BuddyTestRunner は次の要領でテストを実行する。まず第 3 引数で与えられた関数を実行してテストを定義する。次に、定義されたテストが Jasmine により逐一実行される。最後に、テストの実行結果を表すプロミス（Promise オブジェクト）が BuddyTestRunner の戻り値として返される。

例として、スクリーンプログラムにおいてボタン BUTTON1 が押されたら前節の myTest 関数を呼び出してテストを実行し、返されたプロミスの then メソッドを通じてテストの実行結果を受け取るプログラムを以下に示す。

```
module.exports = function(api){
	var actions = {

	};
	var events = {
		BUTTON1_onClick: function(evt) {
			myTest(api, this).then(function (result) {
				return new Promise(function (resolve, reject) {
					console.log(result.status); // 終了状態を表示
					console.log(result.logs.join('\n')); // 実行経過のログを表示
					resolve(result.status);
				});
			});
		},
		// （後略）
	};
};
```

then メソッドに与えた関数の引数には BuddyTestRunner の実行結果オブジェクトが渡される。実行結果オブジェクトは以下のプロパティから成る。

- status: 終了状態を表す文字列。"passed" ならばすべてのテストが成功、"failed" ならば一つ以上のテストが失敗したことを表す。
- logs: テストの実行経過のログを表す文字列の配列。

上記のコード例では myTest 関数に引数としてスクリーンプログラムの api オブジェクトと名前空間 this を渡しており、myTest 関数ではそれらを api 引数および scope 引数として受け取っている。そのため、例えばスクリーン中に配置されたスクリーンモジュールの配列（this.items）は myTest 関数内に記述するテストの中では scope.items として参照できる。DB テーブルの配列（this.tables）、DB ビューの配列（this.views）などについても同様である。

4 サンプルプログラム

BuddyTestRunner によるテストの具体例として、DB テーブルの入出力操作に関するテストの例を完全なスクリーンプログラムの形で省略せず以下に示す。

```
function myTest(api, scope) {
	// DB テーブル table1 を初期状態に戻す
	function restoreTable1(done) { // ①
		var tableName = "table1";
		api.lib.async.series([function (done) { // ②
			// 全レコードを削除
			var options = {
				where: {ID: {op: '>', value: 0}},
			};
			scope.tables[tableName].deleteRecord(options, function(err, res) {
				if (err) console.log(err.response.text);
				expect( err ).toBeNull();
				done();
			});
		}, function (done) {
			// レコード 2 件を挿入
			var testData = [
				{name: 'Taro', age: 30},
				{name: 'Hanako', age: 20},
			];
			api.lib.async.series(testData.map(function (v) {
				return function (done) {
					var options = {
						data: v,
					};
					scope.tables[tableName].insertRecord(options,
function(err, res) {
						if (err) console.log(err.response.text);
						expect( err ).toBeNull();
						done();
					});
				};
			}), function (err, res) {
				if (err) console.log(err);
				// エラーがないことを確認
				expect( err ).toBeNull(); // ③
```

```
                        done();
                    });
                }], function (err, res) {
                    done();
                });
        }
        var options = {
            timeout: 60000, // 1 分
        };
        return BuddyTestRunner(this, options, function() { // ④
            // readData についてのテスト群を記述
            describe("readData", function () {
                // 各テストに先立って呼び出される前処理を指定
                beforeEach(function (done) { // ⑤
                    restoreTable1(done);
                });
                // テストを記述
                it("returns all records when used unconditionally", function (done) {
                    var tableName = "table1";
                    var options = {};
                    scope.tables[tableName].readData(options, function(err, data) {
                        if (err) console.log(err.response.text);
                        expect( err ).toBeNull(); // ⑥
                        expect( data.length ).toEqual(2);
                        done();
                    });
                });
            });
        });
}

module.exports = function(api){
        var actions = {

        };
        var events = {
            BUTTON1_onClick: function(evt){
                myTest(api, this).then(function (result) {
                    return new Promise(function (resolve, reject) {
                        resolve(result.status);
                    });
                });
            },
```

```
                onLoad: function(){

                },
                onUnload: function(){

                },
        };
        var formulas = {

        };

        return {
                "actions": actions,
                "events": events,
                "formulas": formulas,
        };
};
```

前掲のテストは name カラム（文字列）と age カラム（数値）から成る DB テーブル table1 が定義されていることを想定している。テストプログラムの内容を以下に述べる。まず、myTest 関数の中で table1 の内容を初期状態に戻すためのヘルパー関数 restoreTable1 を定義している（①番のコメントの行を参照）。このヘルパー関数の中では、非同期実行される deleteRecord 関数（レコード削除）と insertRecord 関数（レコード挿入）を api.lib.async.series 関数を用いて逐次実行している（②番）。また、これらの DB 操作が正常終了したことを Jasmine のテスト構文で確認している（③番）。続いて BuddyTestRunner の第 3 引数に与えた関数オブジェクトの中で readData 関数（レコード読み出し）のテストを記述している（④番）。各テストの前に自動的に呼び出される前処理を Jasmine の beforeEach 構文で記述しており（⑤番）、その中で上述のヘルパー関数を呼んでいる。また、readData を選択条件なしで実行してエラーが生じないこと、得られるレコードが 2 件であることを確認している（⑥番）。

5 おわりに

本文書では Buddy アプリの中で Jasmine の構文を用いてテストを記述し、BuddyTestRunner を通じてテストを実行する方法について述べた。

テスト実行環境である BuddyTestRunner は一つの関数として実現されており、テスト群の定義、各テストの実行、テスト結果の表示がこの順で実行されて完了する。したがって、本文書で述べたテスト方法は DB テーブルや DB ビューの入出力操作、サーバー機能の呼び出しなどの逐次的なデータ処理のテストに向いている。

一方、スクリーン設計で作成されるイベント駆動型のユーザインタフェースに関しては、DOM ツリーに基づく静的な見た目についてのテストは比較的容易に記述できるのに対して、ユーザによるボタン操作や文字入力、それらのイベント前後の状態の変化といったユーザインタフェースの動的な側面についてのテストを記述するのは難しい。Buddy アプリのユーザインタフェースの動きを簡便にテストするためのツールを提供することは今後の課題である。

以上