

# The Jasmine Effect

Automate; Don't Panic.

# Jasmine's Got Your Back



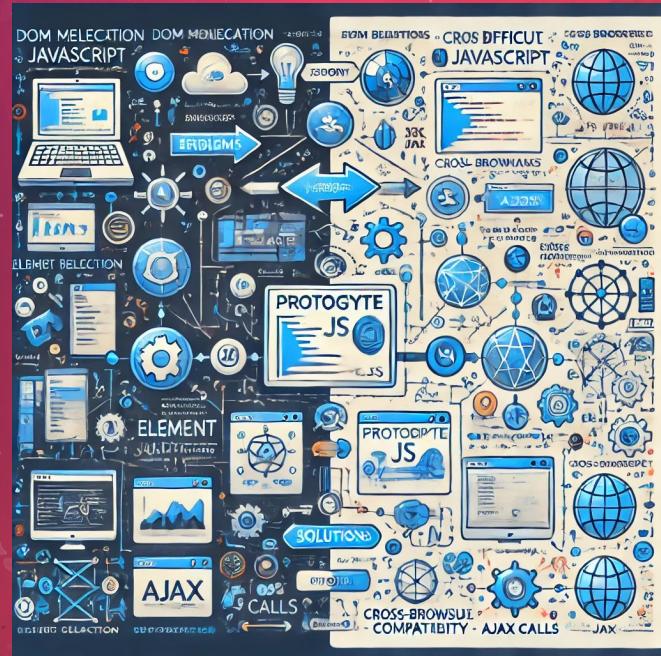
- Jasmine is a lightweight testing framework for testing JavaScript code.
- It does not depend on any other JavaScript frameworks.
- It runs in browsers and in Node.js.
- And it has a clean, obvious syntax so that you can easily write tests.
- Think of it as a detective who catches bugs before they cause trouble.
  - Automated, consistent testing.
  - Catching bugs early in development.
  - Improving script quality and maintainability.

# Prototype

Created by **Sam Stephenson** in 2005 to simplify and improve the process of writing JavaScript for web development.

It provided various utilities and extensions to the JavaScript language, such as:

- Extending built-in JavaScript objects like Array, String, and Function.
- Easier DOM traversal and manipulation.
- Cross-browser support.



# ServiceNow's Secret Sauce

Enter Prototype.js Code

```
var Person = Class.create();
Person.prototype = {
    initialize: function(name) {
        this.name = name;
    },
    say: function(message) {
        return this.name + ':' + message;
    }
};

var guy = new Person('Miro');
guy.say('hi');
```

Miro: hi

The screenshot shows a ServiceNow interface with a code editor and a log panel. The code editor contains the same Prototype.js code as the first panel. The log panel at the top shows the output "Miro: hi". Below the code editor is another code block with two lines of code: `var guy=new Person("Miro");` and `gs.info(guy.say("hi"));`. At the bottom, a message box displays the result: "\*\*\* Script: Miro: hi".

```
1 var Person = Class.create();
2 Person.prototype = {
3     initialize: function(personName) {
4         this.personName = personName;
5     },
6     say: function(message) {
7         return this.personName + ':' + message;
8     }
9 };
10 type: 'Person'
11 };

1 var guy=new Person("Miro");
2 gs.info(guy.say("hi"));

*** Script: Miro: hi
```

# ServiceNow's Secret Sauce

## Enter Prototype.js Code

```
var Person = Class.create();
Person.prototype = {
  initialize: function(name) {
    this.name = name;
  },
  say: function(message) {
    return this.name + ': ' + message;
  }
};

var Pirate = Class.create();

// inherit from Person class:
Pirate.prototype = Object.extend(new Person(), {

  // redefine the speak method
  say: function(message) {
    return this.name + ': ' + message + ', yarr!';
  }
});

var john = new Pirate('Long John');
john.say('ahoy matey');
```

Long John: ahoy matey, yarr!

```
① 
1 var Pirate = Class.create();
2 Pirate.prototype = Object.extendsObject(Person, {
3
4   say: function(message) {
5     return this.personName + ': ' + message + ', yarr!';
6   },
7
8   type: 'Pirate'
9 });
```

```
1 var john = new Pirate('Long John');
2 gs.info(john.say("ahoy matey"));
```

\*\*\* Script: Long John: ahoy matey, yarr!

# Script Includes



- Script Includes are reusable server-side script logic that define a function or class.
- Script Includes execute their script logic only when explicitly called by other scripts.
- They are a powerful way to write code that can be called from multiple places, making your instance more modular, maintainable, and efficient.

# Script Includes Patterns

```
1 var ParentSI = Class.create();
2 ParentSI.prototype = {
3     initialize: function() {
4         gs.info("initialize: ParentSI method.");
5     },
6     parentFunc: function() {
7         gs.info("parentFunc: ParentSI method.");
8     },
9     funcToBeOverriden: function() {
10        gs.info("funcToBeOverriden: ParentSI method.");
11    },
12
13     type: 'ParentSI'
14 };
```

```
1 function sumTwoNums(int1, int2) {
2     var mySum = int1 + int2;
3     gs.info("The sum of " + int1 + " + " + int2 + "=" + mySum);
4 }
```

# Script Includes Patterns

```
1 var Person = Class.create();
2 Person.prototype = {
3     initialize: function(personName) {
4         this.personName = personName;
5     },
6     say: function(message) {
7         return this.personName + ':' + message;
8     },
9     type: 'Person'
10};
```

```
1 var Pirate = Class.create();
2 Pirate.prototype = Object.extendsObject(Person, {
3     say: function(message) {
4         return this.personName + ':' + message + ', yarr!';
5     },
6     type: 'Pirate'
7});
```

# Script Includes Patterns

```
1 var ChildSI = Class.create();
2 ChildSI.prototype = Object.extendsObject(ParentSI, {
3
4     initialize: function() {
5         gs.info("initialize: ChildSI method.");
6         ParentSI.prototype.initialize.call(this);
7     },
8
9     childFunc: function() {
10        gs.info("childFunc: ChildSI method.");
11    },
12     funcToBeOverriden: function() {
13        gs.info("funcToBeOverriden: ChildSI method.");
14    },
15
16
17     type: 'ChildSI'
18 });

```

```
1 var ConstantsSI = Class.create();
2 ConstantsSI.myConstVar = "myConstVar";
3 ConstantsSI.prototype = {
4     myVar: "myVar",
5     initialize: function() {},
6     myFunc: function() {
7         gs.info("myFunc");
8     },
9
10    type: 'ConstantsSI'
11 };
12 ConstantsSI.prototype.varFunc = function() {
13    gs.info("varFunc");
14 };
15 ConstantsSI.constFunc = function() {
16    gs.info("constFunc");
17 };

```

# Run Server Side Script

Test Step  
Run Server Side Script

Execution order  Application Global 

Active  Test JasmineTest 

Step config Run Server Side Script 

Description Run Server Side Validation Script

Notes 

\* Jasmine version 3.1 

Test script   Turn on ECMAScript 2021 (ES12) mode 

```
1 // You can use this step to execute a variety of server-side javascript tests including
2 // jasmine tests and custom assertions
3 //
4 //
5 // Pass or fail a step: Override the step outcome to pass or fail. This is ignored
6 // by jasmine tests
7 //
8 // - Return true from the main function body to pass the step
9 // - Return false from the main function body to fail the step
10 //
11 //
12 // outputs: Pre-defined Step config Output variables to set on this step during
13 // execution that are available to later steps
14 //
15 // steps(SYS_ID): A function to retrieve Output variable data from a step that executed
16 // earlier in the test. The desired step's sys_id is required
17 //
18 // params: The current test run data set's parameter data including both
19 // exclusive and shared parameters
20 //
21 // Example:
22 //
```

# Suites: describe Your Tests



- The `describe` function is for **grouping related specs**; typically, each test file has one at the top level.
- The string parameter is for naming the collection of specs, and will be concatenated with specs to make a spec's full name.
  - This aids in finding specs in a large suite.
  - If you name them well, your specs read as full sentences in traditional BDD style.

# Specs



- Specs Specs are defined by calling the global Jasmine function `it`, which, like `describe` takes a string and a function.
- The string is the title of the spec and the function is the spec, or test.
- A spec contains one or more expectations that test the state of the code.
  - An expectation in Jasmine is an assertion that is either true or false.
  - A spec with all true expectations is a passing spec.
  - A spec with one or more false expectations is a failing spec.

# Expectations



- Expectations are built with the function `expect` which takes a value, called the actual.
- It is chained with a Matcher function, which takes the expected value.

# Matchers



- Each matcher implements a boolean comparison between the actual value and the expected value.
- It is responsible for reporting to Jasmine if the expectation is true or false.
  - Jasmine will then pass or fail the spec.
- Any matcher can evaluate to a negative assertion by chaining the call to expect with a **not** before calling the matcher.
- Jasmine has a rich set of matchers included, you can find the full list in the API docs.

# Setup and Teardown



- To help a test suite **DRY** up any duplicated setup and teardown code, Jasmine provides the global `beforeEach` , `afterEach` , `beforeAll` , and `afterAll` functions.
- As the name implies, the `beforeEach` function is **called once before each spec** in the describe in which it is called and the `afterEach` function is **called once after each spec** .
- The `beforeAll` function is **called only once before all the specs** in describe are run, and the `afterAll` function is **called after all specs finish** .
- `beforeAll` and `afterAll` can be used to speed up test suites with expensive setup and teardown.

# The this keyword



- A way to **share variables** between a `beforeEach`, `it`, and `afterEach` is through the `this` keyword.
- Each spec's `beforeEach/it/afterEach` has the `this` as the same empty object that is set back to empty for the next spec's `beforeEach/it/afterEach`.
- **Note:** If you want to use the `this` keyword to share variables, you must use the **function keyword** and not arrow functions .

# Disabling Suites



- Suites can be disabled with the `xdescribe` function.
- These suites and any specs inside them are skipped when run and thus their results will show as pending.
- Suites can also be focused with the `fdescribe` function.
  - That means only fdescribe suits will run.

# Pending Specs



- Pending specs do not run, but their names will show up in the results as pending.
- Any spec declared with `xit` is marked as pending.
- Any spec declared without a function body will also be marked pending in results.
- And if you call the function `pending` anywhere in the spec body, no matter the expectations, the spec will be marked pending.
  - A string passed to pending will be treated as a reason and displayed when the suite finishes.
- Tests can also be focused with the `fit` function.
  - That means only fit tests will run.

## Manually failing a spec with fail



- The fail function causes a spec to fail.
- It can take a failure message or an Error object as a parameter.

# Spies



- Jasmine has test double functions called `spies`.
- A spy can stub any function and tracks calls to it and all arguments.
- A spy **only exists in the describe or it block in which it is defined**, and will be removed after each spec.
- You can define what the spy will do when invoked with `and`.
- There are special matchers for interacting with spies.
  - The `toHaveBeenCalled` matcher will pass if the spy was called.
  - The `toHaveBeenCalledTimes` matcher will pass if the spy was called the specified number of times.
  - The `toHaveBeenCalledWith` matcher will return true if the argument list matches any of the recorded calls to the spy.

# Spies: createSpy



- When there is not a function to spy on, `jasmine.createSpy` can create a "bare" spy.
- This spy acts as any other spy - tracking calls, arguments, etc; But there is no implementation behind it.

# References

- <http://prototypejs.org/>
- <https://beingfluid.github.io/ptototypejstest/>
- [Objects: Making It A Little Easier...](#)
- [Script Includes](#)
- [NOWCommunity Live Stream - Code Decoded - Using a Constants script include](#)
- [Live Coding Happy Hour for 2017-02-10 - Server Scripts with Automated Test Framework](#)
- [Unit testing your Javascript with jasmine](#)
- <https://jasmine.github.io/>
- [Unlearn Series - Unit Test Your Script Include Using ATF](#)
- [Learning Unit Testing with Jasmine](#)
- <https://beingfluid.github.io/ptototypejstest/code.html>